The Design and Evaluation of a Multiple–Language Active Network Architecture Enabled via Middleware

---

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science in Computer Science
in the
University of Canterbury
by
Carl Cook

Examining Committee

Associate Professor Krzysztof Pawlikowski	Supervisor
Associate Professor Harsha Sirisena	Associate Supervisor

University of Canterbury
2001

To Debby, Colin, Benjamin, and Paige

Abstract

In conventional data communication networks, the basic network components are passive; routing decisions are made solely on the basis of packet header information. In contrast, active networks allow added computation within the network through user-defined routing and processing instructions, providing the on-demand installation of powerful software-based network services.

As an adaptation of previous active networks, this thesis presents an architecture based entirely in middleware. By utilising middleware services, the architecture resolves authentication, memory-management, and interconnectivity issues otherwise assumed as inherent, and enables a highly functional multiple-language interface for the deployment of dynamic protocols. After describing the architectural design, an empirical system evaluation is presented with comparisons to both conventional network protocols and a well-known existing active network architecture. Results indicate performance improvements over the existing architecture, and demonstrate the feasibility of a multiple-language active network infrastructure implemented entirely in middleware.

Table of Contents

Acknowledgments

I would like to thank my supervisors Associate Professor Kryztov Pawlikowski and Associate Professor Harsha Sirisena for their patience, guidance, and enthusiasm as this research project and thesis grew from a late night burst of inspiration to a reality.

I also would like to acknowledge the University of Canterbury's department of Computer Science for providing a quality research environment, in particular through the knowledgeable and supportive academic staff. Additionally, I would like to thank the University of Canterbury for providing me with the resources to fulfil this research. In similar fashion, I also appreciate the generous support from the Christchurch office of Trimble New Zealand Limited.

Finally, I would also like acknowledge the support of my colleagues: André, Andreas, Enoch, Michael, Pramudi, Theuns, and Tim. Whilst far too many days were spent collectively considering the commercial viability of waterproof teabags, the violent displays of panic by my colleagues as their own deadlines approached provided me with ample motivation to complete this work on time.

Chapter I

Introduction

1.1 Definition of an Active Network

Components within conventional IP-based networks are passive in that routing decisions are made solely on the basis of packet header information. Packet payloads are not processed within the network, thus limiting network functionality and capabilities. Some end-to-end services such as hypertext can be provided within the existing IP infrastructure of simple packet forwarding, but the current surge in demand for network-level functionality (as reflected by Internet Engineering Task Force activities on IntServ/RSVP, DiffServ, and IPv6) suggests that the current 'one size fits all' IP model may no longer be suitable for today's networks.

The new *active network* paradigm, in contrast, provides networked applications and users with a programmable interface that supports the dynamic modification of a network's behaviour—bypassing both the standards committee and the hardware vendor in order to cater for ever-changing user demands. Such flexibility is achieved by allowing the specification and activation of complex processing instructions at all participating intermediate active network routers, facilitating the run-time installation of arbitrary software-based routing protocols [49]. Numerous applications of active networks may be envisioned, such as dynamic compression, arbitrary network caching, and on-demand encryption between endpoints in conventional, multi-casted, and future IP-based networks. Whilst several implementations of active network architectures exist within the network research community, active networks are not yet being used commercially.

1.2 Background

The DARPA Information Technology Office sponsors and governs the main body of active network research. In line with the previous definition of an active network, DARPA's mission for all active network research is to "enable networks that turn on a dime" [30]. Active network research is primarily concerned with providing efficient and highly flexible multi-layered networks that do not compromise the security of end-users, intermediate hosts, or the network state itself. The remainder of this section expands on the primary goals of active network research.

1.2.1 Functionality

An important goal of active networks is to markedly increase the level of functionality that is currently offered by conventional networks. By offering virtually unbounded levels of routing computation within the network itself, it is envisaged that new services can be deployed rapidly without the need for universal standardisation.

It should also be possible for active networks to keep pace with the rapid increases in network complexity as user demand continues to expand. As network topologies and throughput rates grow, it is expected that the active network paradigm will be able to address the current anomaly between the rate at which users' demands change and the pace at which new services can be deployed.

1.2.2 Efficiency

An unprecedented level of node computation is available to an active network when compared to the packet processing power of conventional networks. Nodes within an active network can exploit the current network conditions and other network state information to gain optimisations that are otherwise considered unobtainable.

By introducing the run-time verification of network states prior to making routing decisions, it is theoretically possible to eliminate the need for packet retransmission. This is achieved by providing network logic that can simply

reroute congested network paths and bypass links with high error rates. An appropriate analogy can be drawn from a commuter selecting an alternative route to the destination because of a traffic report broadcasted on the car radio. Hence, a 100% increase in the useful data rate to networked applications is a realistic goal for active network architectures.

1.2.3 Safety and Security

The active network paradigm strives to increase the level of security available in conventional networks by recognising a multi-tiered structure of users and services for both static and mobile networks. A fundamental principle of active networks is that now both users and processes require authentication to allow safe access to network resources. Subsequently, separate data transmission and administrative functions must be controlled according to the type or request and the security policy in place [3].

1.3 Outstanding Active Network Issues

Many aspects of active networks remain unresolved, and consequently the first generation of active network research appears to be fading out [33]. Such issues include the high level of performance degradation at each active node, the security risks of processing arbitrary code modules, and the establishment of bounds for core active network services. Whilst the provision of enhanced security services is an important goal of active network research, it must also be stated that the preservation of existing network security is a goal within itself. Allowing the introduction of arbitrary routing computation on intermediate nodes poses many new security issues, in terms of both network stability and the protection of active hosts.

To elaborate on node performance issues, it is apparent that active nodes in a network perform a greater level of processing per packet. Depending on the routing logic installed in the active network, active nodes may also have considerable overhead inflicted upon them in order to maintain state information such as traffic flows and error rates. Whilst there can be a marked per-node degradation in performance when comparing the role of packet for-

warding on an active network to a conventional network, it is expected that the inter-node optimisations such as congestion avoidance and intelligent data caching can improve the overall rates of end-to-end communication. However, this is still to be proven for many network environments.

1.4 Research Goals

This thesis aims to provide:

1. A comprehensive survey of current active network research including existing architectures, active network-specific languages, host implementation languages, and specialised protocols and interfaces.

2. An analysis of middleware-based active network architectures, leading to the design of a functional middleware-based active network prototype.

3. A detailed performance analysis of the active network prototype under various topologies.

Results from this research will be used to aid the future development of active network architectures, protocols, and applications by verifying current assumptions and/or exposing any anomalies that may be present within the paradigm of current active networks. After assessing the feasibility of the active network prototype, as the basis of this research, it is envisaged that a hardware-based active router could be developed with the aim to provide high-performance next-generation networks without sacrificing network security.

1.5 Thesis Structure

Chapter 2 of this thesis introduces the active network paradigm with a survey of the various architectures and languages that are associated with the first generation of active networking research. Next, Chapter 3 reveals the outstanding issues such as performance degradation and the security risks of processing arbitrary code modules.

Chapter 4 of this thesis analyses the role of middleware in active networking, by explaining the origins of middleware and revealing its suitability as an implementation framework for an active network architecture. This chapter then presents the Component Object Model Active Network (COMAN) architecture and explains the motivation for such an implementation. The design principles and implementation details of the architecture are also described. Chapter 5 presents details of an evaluation of COMAN's performance, including both experimental design and results. Outstanding issues and limitations of the architecture are discussed in Chapter 6, providing direction for further work.

Chapter II

Previous Active Network Research

This chapter surveys the current state of active networks and associated research projects. The motivations for the development of active networks are discussed, followed by a description of a generic active network architecture. Following this, several current active network architectures are presented, including a description of several frameworks that address active network security. Next, active network specific programming languages are discussed, followed by a presentation of several individual active network protocols and interfaces. Today's common programming languages are then addressed to reveal the advantages and disadvantages of software-based active networks. Finally, recent work towards testbed networks for active network research is introduced.

2.1 Motivations for an Active Network

As noted by Tennenhouse and Wetherall [49], an object-oriented approach to networking is suggested every five or ten years, with varied levels of success. Two early examples of object-oriented network services are SNMP v2.0 [13] and the ACE socket wrappers [44]. SNMP v2.0 introduced an object-oriented management information base to better contain collections of related data modules, which quickly spawned several proprietary implementations of an object-oriented API named SNMP++. Similarly, the ACE socket wrapper implementation borrowed some of the high-level concepts from the object-oriented programming paradigm, providing a set of socket classes that remove many of the low-level complexities and sources of potential application-programmer errors.

Whilst the above examples display that an object-oriented approach to

networking can provide benefits in terms of managing programmatic complexity, the main goal of conventional networks has always been that of performance. As the IP protocol has shown us for the last decade, performance can be achieved by limiting processing wherever possible. By restricting configuration to only a handful of low-level IP primitives such as the *type of service* field, network routers are allowed to concentrate their processing on what IP does best—that of storing and forwarding packets.

From a vastly different perspective, the drive for greater performance of software systems threw the entire computing industry (in particular the financial sector) into a crisis for the majority of the 1980s. Analysis of the aftermath revealed that the growing complexity of software systems greatly increased the number of errors and development time for not just the modification and expansion of projects, but in routine system maintainance itself. The threat of unmanageable software systems due to issues of complexity spawned a shift in the programming paradigm to that of object-oriented software development—which was up until this time considered by the industry to be a novel idea but a gross waste of resources.

Under the new paradigm, software engineers could reuse 'known good' code components, or adapt components for specific needs via inheritance rather than recreating entire systems from scratch. By using an object-oriented (or component-based) approach, complexity could be managed by logically dividing the entire system into subsystems for development by individual project teams. As an analogy, this is similar to the way that electrical engineers select basic components such as resistors and capacitors for the construction of new devices, rather than rebuilding the entire device. Subsequently, software development in the 1990s proved the concept of object-oriented programming, with industries' most popular language now being C++. Other object-oriented languages such as Java are also gaining a strong following.

The initial resistance to object-oriented software development was mainly due to the price paid in terms of system performance. However, the actual cost for object-oriented methodologies and languages is in nearly all cases insignificant because of the ever-reducing cost of computing power, not to

mention the above benefits such as component reuse and management of complexity. Hence, the object-oriented programming paradigm has, over the years, defied its label as an 'inefficient' practice.

As the Internet continues to expand at an exponential rate of growth in terms of both traffic volumes and number of hosts, the software crisis of the 1980s should act as a constant reminder to control all associated complexities in networking or suffer the consequences such as decreased performance and slow adaptability. The drawn-out nature of IPv6 adaptation provides evidence that the complexity of the Internet is already raising managability concerns. Even many private IP networks suffer from issues of complexity, especially where multimedia traffic is involved, as indicated by the ever increasing popularity of media-stream servers, application-level firewalls, and caching proxy servers—all catering for functions that the IP protocol overlooked and does not scale well to.

Accordingly, this is where the role of active networks comes into existence. As introduced by Saltzer et al. [43], the *end-to-end argument* in the context of active networks suggests that performance should be measured in terms of application layer throughput, rather than in pure transport or network layer performance. Active networks allow vastly increased levels of computation at potentially every node in a given network, allowing the run-time adaptation of new protocols and services through comprehensive software interfaces, as opposed to hardwired IP networks that only allow minimal changes in configuration.

Today's networks are becoming unmistakably more complicated as both the size and number of host services increases. Any changes to networks, ranging from protocol version updates to the introduction of new protocols, require either service interruptions or specialised bridging hardware and software (which is then discarded), with both alternatives being costly in terms of both time and resources. Additionally, whenever changes are introduced, new components and services tend to be constructed from the ground up, using IPv6 as a relevant example.

Current research in active networks suggests that as network researchers, we could learn from the software crisis, and avoid most of the problems

that the software industry of the late 1980s incurred—problems such as a vast increase in complexity due to a low-level and unstructured approach to building systems. By reusing existing components such as 'configurable' base network services, and by subsequent development and deployment of higher-layered routing protocols on demand (in software and/or hardware), it may be possible to address the anomaly between changing user demands and the current time taken to upgrade networks, regardless of the underlying transport mechanism.

Additionally, by giving a degree of network service control to end applications, the underlying network logic can afford to give less consideration to the type of data being transferred. For example, where a multimedia stream is being transferred, particular attention must be paid to the order of packet delivery, as well as to satisfy a pre-specified minimum throughput rate. However, active networks allow the data itself to specify how it is to be processed at each point in the network.

The obvious tradeoff for active networks is in terms of performance, but as the cost of computational power within nodes continues to decrease, the subsequent increase in control over network processing allows the network to optimise traffic flows, depending on rates of congestion and user-defined parameters. Additionally, new services can be introduced such as on demand compression and encryption, multicasting, QoS monitoring and enforcement, and mobile host adaptation. Following the end-to-end argument, current research into active networking attempts to prove that such enhancements to delivery of data at the application level results in an overall improvement in network utilisation, for a modest increase in the computational power required for network devices.

2.2 General Active Network Architectures

This section provides an introduction to the typical active network architectures, presenting the components from which such active networks are constructed. Active network architectures tend to fall into two broad categories: *packet-based* and *node-based.* An explanation of these categories is now presented, followed by an overview of the individual active network

components.

2.2.1 Packet-Based Active Networks

Packet-based, or capsule-based, active networks are arguably the most radical shift away from conventional networks, as individual packets in a packet-based network carry the router processing instructions inline. Not only can every packet in a stream contain different processing and routing instructions, the logic of these instructions can result in packets duplicating themselves for the purposes of multicasting [10], or even changing a packet's own routing logic via dynamic programming. One packet-based active network project, named Cognitive Packet Networks, extends as far as using machine learning algorithms to dynamically route TCP packets [19].

The underlying principle of all packet-based active networks is the same: provide a relatively small unit of code from a known language within each packet header during packet construction, and all active nodes within the network will execute the instructions accordingly. A secondary role of active network nodes could be the maintainance of a soft information state per node, and/or the concatenation of additional data or processing instructions onto specific packets. Packet processing instructions are generally in the form of a safe scripting language such as SafeTcl [37], or an intermediate language such as byte-compiled Java. The merits of these data processing languages and others are discussed in Section 2.5.

2.2.2 Node-Based Active Networks

Somewhere between the radical realms of packet-based active networks and the more traditional IP-based networks lie node-based active networks. Node-based active networks allow the run-time specification of *code modules*, but they are installed at active nodes on the network as opposed to being carried inline by individual packets. Packet processing is initiated at active nodes when the inspection of a packet reveals a reference to a known code module. Because node-based active networks are not subjected to the overhead of carrying routing instructions for every packet of transmitted data, installed

routing modules tend to be larger and more complex, offering a potentially unbounded array of routing and processing functions.

According to the constraints of the given node-based architecture, the end-programmer associates code modules to either an entire stream of data or individual packets—normally via the architecture's API if available. This association is made by inserting code module references into the packet or stream header, with a lookup function at each active node proving the link between the packets and their associated code modules. Hence, node-based active networks are very similar to conventional IP-based networks, with the exception of unbounded user-specified processing and routing logic at every point within the network.

2.2.3 Active Network Components

Regardless of whether an active network is node-based or packet-based, active networks tend to share the same fundamental construction. Figure 2.1 presents this general active network architecture, which is composed of several base components. Whilst this figure represents a simplification of current active network components and architectures, all of the architectures and components presented in this chapter can be classified by this model.

A simplistic version of the architecture presented herein is provided by the Active Network Working Group's recent draft RFC entitled *Architectural Framework for Active Networks Version 1.0* [11]. The purpose of this Internet draft is to identify the core set of active network components and services prior to standardisation. Accordingly, active network architectures can be represented by the following general components:

- Active applications
- Packet languages (optional)
- Execution environments, which consist of:
  - An active network daemon/service
  - An implementation language

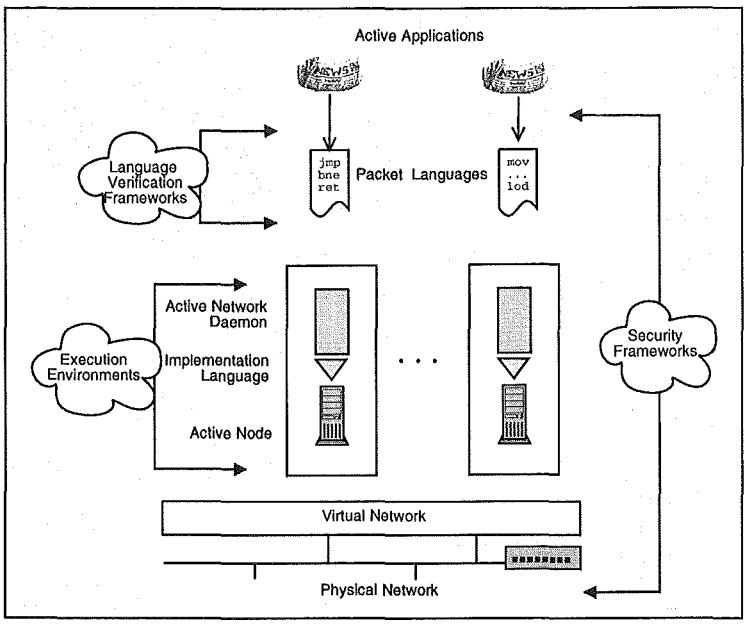

Figure 2.1: A general model of the core active network components.

- A physical network host/active node

- A virtual network (optional)

- A physical network

Additionally, frameworks exist to formally verify the safety of packet languages. Similarly, frameworks also exist to ensure the security of an entire active network, ranging from bootstrapping the active nodes to authenticating packets of data. Finally, several protocols and published interfaces are defined to facilitate communication between the components of an active network.

In the remaining sections of this chapter, the above components and frameworks are discussed with reference to relevant implementations.

2.3 *Existing Architectures*

This section introduces the architectures that form the basis for current active network research. As will be revealed, some architectures address the primary goals of active network research, such as functionality, performance, and security, whilst others focus on one specific area. All of the architectures listed are part of the DARPA active network research body, and are presented in order of the first known publication date.

For the majority of the architectures discussed in this section, few objective details have been given by the respective authors regarding the type and bounds of the equipment used and the underlying network structure. This leaves many characteristics of the architecture unresolved, particularly in terms of performance and security. Subsequently, the issue of achieving an objective and thorough evaluation of an active network architecture forms the basis for Chapter 3: *Unresolved Issues of Active Networks.*

2.3.1 ANTS

The ANTS active network was introduced in April 1996, and is possibly the most adaptable (if not the most researched) architecture of today [52]. The architecture was initially packet-based [51], meaning that the specialised routing and processing instructions are carried within the header of each packet. The opposite approach is that of node-based architectures, where pre-specified code modules at each active node process the packets that contain relevant reference(s) in the packet header (an example of such a system is presented in Section 2.3.5). However, the latest version of ANTS uses a code module caching system to avoid sending duplicate code modules.

The ANTS architecture is implemented in Java, providing many of ANTS's distributed functions natively. Code modules can be user-defined, which requires an ANTS safety thread to be running constantly in the background, handling any code module that consumes excessive resources, such as CPU time and memory beyond the pre-specified limits. The routing API, as provided by the architecture, offers only a limited set of commands to provide security at active nodes. The Java run-time libraries also attempt to protect

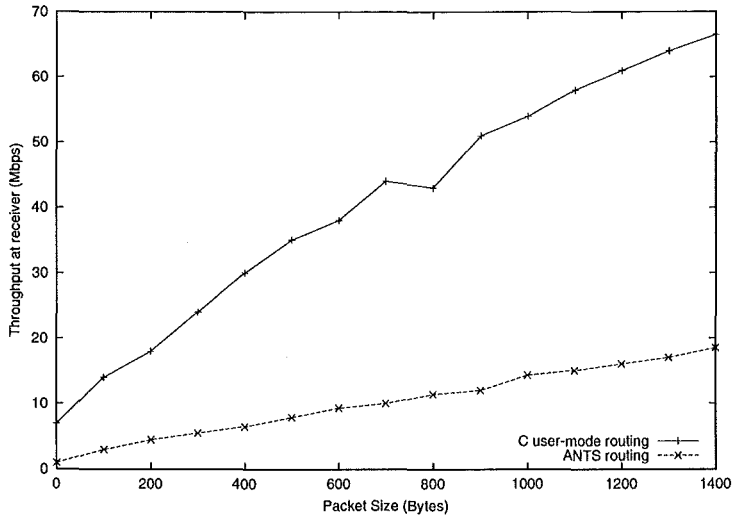

Figure 2.2: The rate of throughput for a single-hop data transfer for the ANTS architecture using various buffer sizes.

out-of-process resources, with the exception of shared memory that may be utilised by all users.

Applications including multicasting [10], mobile host accommodation [28], and congestion control [6] have been implemented using the ANTS architecture, proving the concept that such applications can be implemented using such a framework. However, the latest version of the system (version 2.0) only claims a maximum throughput of 18 Mbps, as presented in Figure 2.2. Additionally, latencies above $700\mu s$ are also experienced when buffer sizes exceed 1,100 bytes, as can be seen from Figure 2.3. Despite the unimpressive performance, ANTS has proven to be a very useful (and hence highly utilised) framework for implementing and testing first-generation active network applications and protocols.

2.3.2 Netscript

The Netscript project, as introduced by Yemini and da Silva [54] in April 1996, focuses primarily on providing a simple programmable network with efficiency as a key priority. This allows the network vendors to install new routing protocols with the same ease as a user installs and runs a new appli-

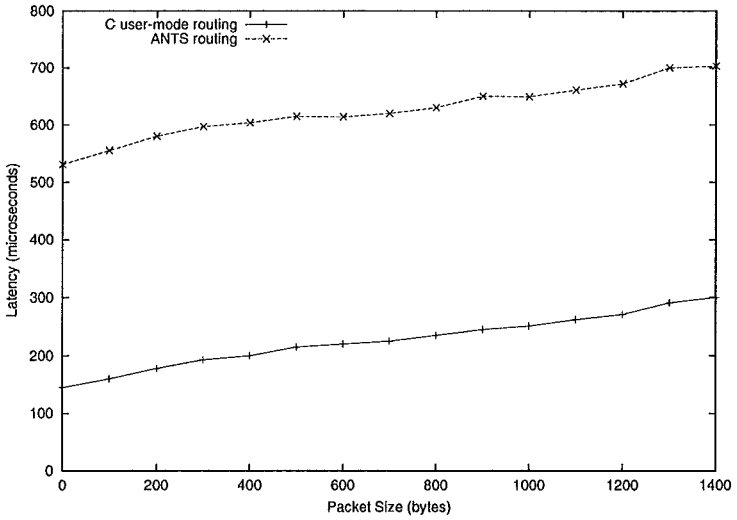

Figure 2.3: The latency of data packets for the ANTS architecture using various buffer sizes.

cation on an end-node. Netscript achieves this goal by providing a minimal set of primitives for *agents* (otherwise known as a dynamically dispatched and remotely executed program) to call in order to accomplish advanced network processing and routing.

The architecture that transports Netscript agents is referred to as a Netscript Virtual Network (NVN). An NVN consists of two entities: Virtual Network Engines (VNEs) and Virtual Links (VLs). Whilst VNEs appear to be analogous to active nodes, and VLs appear to be analogous to physical links between active nodes, the architecture states that several independent VNEs may reside on any given device in the network, maintaining their own separate information bases. Additionally, VLs can also represent any number of physical network links, which is particularly applicable to multicasting. Hence, the composition of an NVN is only loosely coupled with the underlying physical network. As the VEs in the network store and execute the agent scripts, Netscript appears to be a node-based active network architecture.

Due to the restrictive nature of the Netscript language, agents tend to perform only lightweight duties such as network management and external protocol analysis/profiling. Consequently, Netscript provides an ideal alter-

native to traditional means of network management such as remote SNMP monitoring. The volume of traffic generated by remote SNMP makes profiling large networks impractical, whereas Netscript agents can provide an unbounded array of monitoring options without having to probe a remote SNMP Management Information Base (MIB). This concept was later expanded to provide lightweight Netscript agents to implement SNMP functions, ranging from managing the MIB to setting user-requested SNMP traps.

2.3.3 Liquid Software

The Liquid Software project was introduced in November 1996 by Hartman et al. [21]. Instead of immediately developing a new framework, the aim of this research is to develop software and software systems that can easily flow from one node to another, with potentially no additional constraints or modifications to the underlying network and operating systems. It is envisaged that the Liquid Software system will enable active networks rather than implement them. According to the authors, networks built using Liquid Software will be easier to maintain, debug, and update.

Current research by the Liquid Software group focuses on methods to provide portable code, and methods to interpret or compile intermediate code in a just-in-time nature to adhere to performance constraints. An API is currently being developed, which will take into account the large number of operating systems and architectures for which the system will be implemented. It is expected that the API will also take into account common security routines, which the system must enforce. It must be noted that at this stage of the research, the project is investigating the requirements that comprise the core set of API routines, rather than focusing on specific system implementations.

2.3.4 The Switchware Project

Whilst the Switchware project was first presented by Smith et al. [47] in 1997, the paper written by Alexander et al. [2] in May 1998 introduced the main features of the architecture. To address security issues, the Switchware project of May 1998 presented a layered approach to providing a flexible,

yet safe and secure network. Layers in the architecture consist of specialised active packets, active extensions, and a secure active node infrastructure.

As the host language for Switchware's active packets, the PLAN language was created to meet the project's safety requirements (see Section 2.5.1). To implement the secure active node infrastructure, the SANE environment was developed with various features such as a secure boot-loader and the verification of system integrity via cryptographic authentication (see Section 2.4.1). Nested between the active node infrastructure and the PLAN active packets is the active extensions layer. This layer is required to provide additional functionality to the active packets due to the restrictive nature of the PLAN language primitives. Converse to the inline routing instructions of Switchware active packets, more powerful active extensions are installed on specific nodes in the network by an administrator, with access to active extensions controlled by the underlying SANE environment.

The Switchware project employs a language-based approach to maintaining security. In further detail, the concepts of proof carrying code, static and dynamic type checking, and program verification are employed to verify that code modules in both active packets and active extensions have not been altered from their original state. All active nodes in the architecture perform such static and dynamic checks on code modules, as well as performing various other authentication tasks.

To prove the concept of the Switchware architecture, the PlanET environment was created to emulate a medium-sized WAN, consisting of seven nodes and a source-to-destination hop-count of four [24]. It was shown that by using a relatively simple architecture with limited flexibility, end-to-end throughput rates of up to 40 Mbps on a 100 Mbps Ethernet link is possible (as presented in Figure 2.4 using a four-hop network). However, the issue of an increase in latency per hop was not resolved, as is evident in Figure 2.5. Subsequent to these initial trials, new objectives in terms of security, performance, and functionality were defined, which marked the start of development for the SNAP packet language (see Section 2.5.2), with the eventual introduction of the SNAPd active network environment (see Section 2.3.8).

Finally, the Switchware project also introduced the ANEP protocol in the

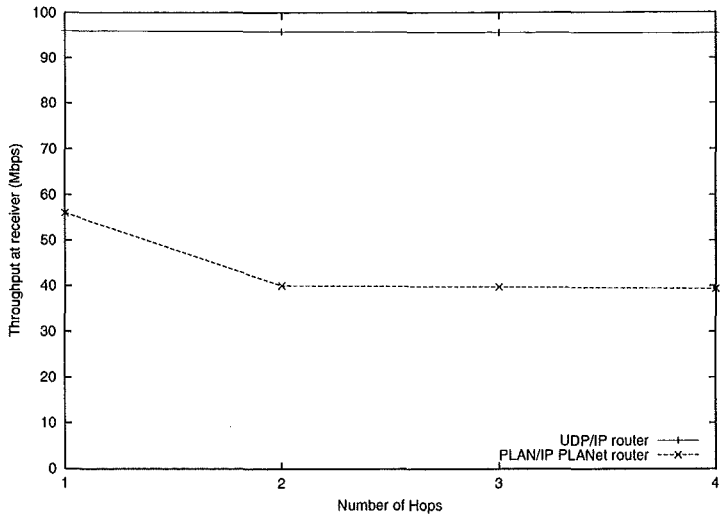

Figure 2.4: The rate of throughput for multiple-hop data transfer for the PlanET architecture using a 1,500 byte buffer size.

form of an RFC, in an attempt to standardise active network research and architectures for future interoperability (see Section 2.7.2 for a description of the ANEP protocol). Subsequently, the ANEP protocol has been used by the BOWMAN node operating system (see Section 2.3.7) and the SmartPackets architecture (see Section 2.3.6) among other more recent architectures.

2.3.5 PAN

The PAN active network implementation was introduced in April 1998 by Nygren [35]. The system is very much a hybrid of the ANTS system with a focus of high performance. If anything, the PAN architecture provides evidence that software based active networks can achieve as little as 13% overhead when forwarding packets over a single active node. However, both functionality and security were compromised to provide such results.

The PAN active network differs little from the standard IP kernel modules that are found in typical operating systems such as Unix and Windows NT. Not only is it highly unsafe to allow user-defined instructions to be executed in kernel mode (as acknowledged by the author), the claim that the system supports multiple code systems is not as convincing as first thought. The

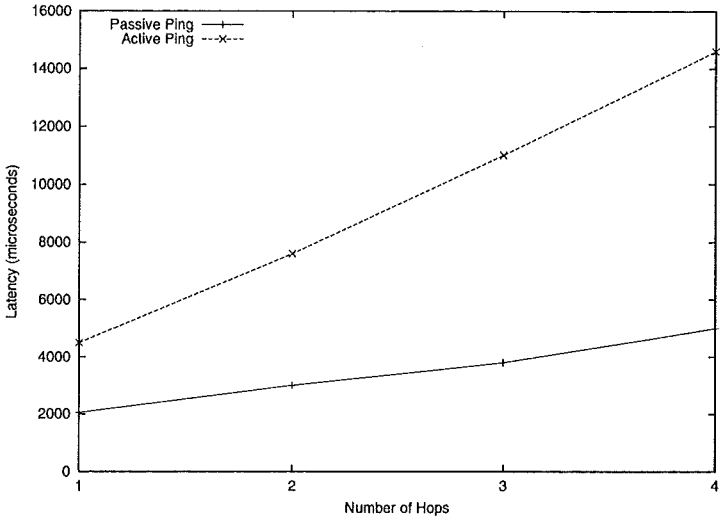

Figure 2.5: The latency of multiple-hop `ping` packets for the PlanET architecture using a 1,500 byte buffer size.

architecture only supports the `a.out` and ELF linking formats, which are specific to Unix and its hybrids. As explained in Section 2.6.3, alternative frameworks such as middleware provide true multiple code systems. Whilst the architecture certainly performs well, the author does not describe what sort of active processing (if any) is possible from the architecture, and there is no mention of specialised routing or processing in the presentation of the architecture's results.

As presented in Figure 2.6, only in kernel mode did the system perform well in terms of data throughput. Performance in terms of low latency rates was also favourable to the kernel implementation of the PAN architecture, as presented in Figure 2.7. As the PAN architecture only performed well in kernel mode, it is difficult to argue that the system is a functional, yet safe, active network implementation—as opposed to simply an enhanced (and fundamentally unsafe) network driver. It is unreasonable to assume that an active network architecture can scale to the speed of the PAN architecture by making a few minor adjustments—it would require a major system redesign with unacceptable security and functionality compromises to match PAN's performance levels.

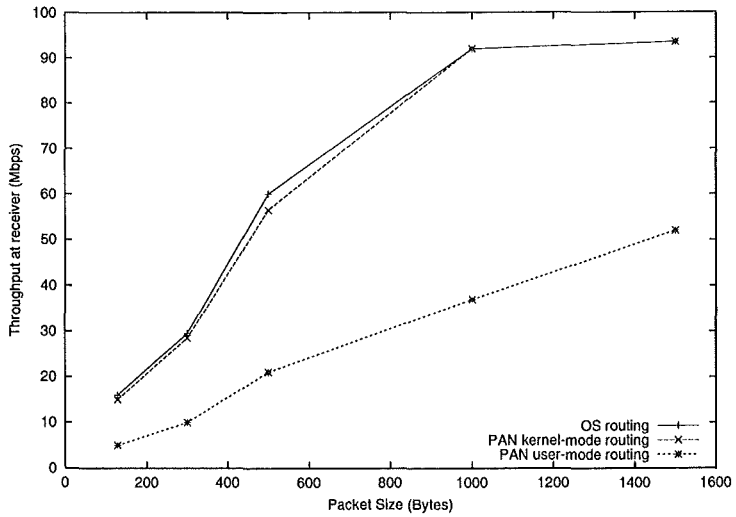

Figure 2.6: The rate of throughput for a single-hop data transfer for the PAN architecture using various buffer sizes.

2.3.6 SmartPackets

Much like Netscript, the SmartPackets architecture focuses on the management of programmable nodes, with particular focus on a lightweight implementation that adheres to strict security considerations. The architecture, as introduced by BNN Technologies in March 1999, demonstrates just how well-suited active networks are to network management.

According to the authors [46], the architecture improves the management of large and complex networks by:

1. Allowing management points to be located closer to the actual node(s) being managed.

2. Retrieving specific characteristics from a node for analysis purposes, rather than performing exhaustive activity polling.

3. Abstracting the concepts of management to actual constructs of the SmartPackets language, allowing fine-tuned network control.

The SmartPackets architecture has demonstrated substantially more power

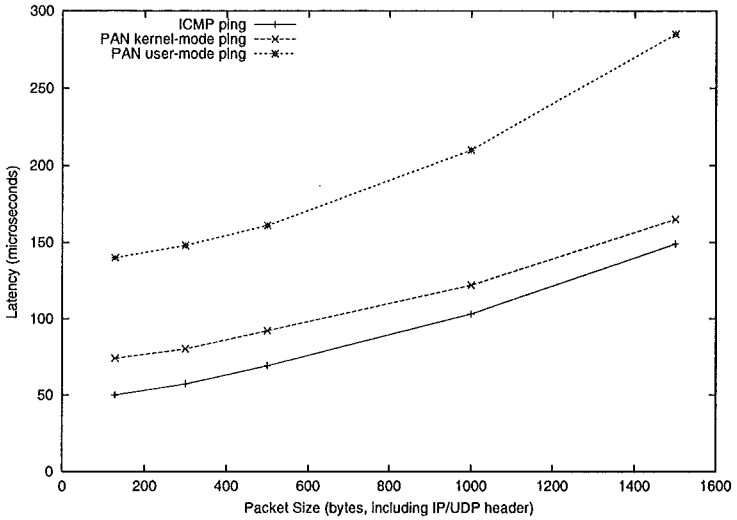

Figure 2.7: The latency of `ping` packets for the PAN architecture using various buffer sizes.

for performing SNMP routines than is possible with conventional implementations. For the manipulation and querying of MIBs, the SmartPacket architecture can provide elegant event-based monitoring to support network management rather than using the brute-force methods of activity polling.

The SmartPackets architecture is constructed from four entities: a specification for the format of smart packets (and their encapsulation into the host network), a specification of the languages that produce the compressed encoding of the executable packet, a virtual machine to provide a context in which SmartPackets are interpreted and executed, and finally, a security model. SmartPackets are encapsulated within the ANEP protocol to allow integration into existing IP networks. Under the SmartPackets architecture, IP routers check the *Router Alert* IP header option and forward ANEP packets to the SmartPackets ANEP daemon, which then hands the packets to the relevant network management program or virtual machine for processing.

Four types of packets exist in the SmartPackets architecture. Program packets carry the executable code to the virtual machines that reside on the active hosts. Data packets carry the result of the subsequent processing back to the caller—which could be a host application or another active node.

Message packets carry pure information, such as MIB entries, as opposed to executable programs or program results. Finally, error packets carry error codes and unhandled exceptions thrown from active nodes upon failure of the network or program execution.

Specialised languages were developed to host SmartPackets instructions. A high-level C-like language called *Sprocket* was implemented to provide familiar and simple program packet development. Most of the C keywords have been retained, but unnecessary constructs, such as typedefs and structures, were removed. A Sprocket compiler is then invoked to output *Spanner* code, which closely resembles assembly code. Of course, Spanner code can be developed by hand, but it is much more likely that any functional Program Packet is first written in the Sprocket language with the resultant Spanner code being hand-optimised where necessary, if performance is the key objective.

Spanner code is interpreted and executed by the virtual machines that reside on the network's active nodes. Execution of such code is considered relatively safe due to the tight set of primitives available, the restriction of memory access to only the in-process stack and local variables (no heap access is permitted), and the lack of dangerous constructs and type-unsafe variables as found in the C language.

The SmartPackets security model provides authentication in the form of cryptographic hashes on all packets, as well as access control lists to restrict the invocation of privileged virtual machine instructions. If any authorisation or authentication event fails, the instructions in the related packet are executed with low privileges to avoid compromising the active nodes or the state of the network.

2.3.7 BOWMAN

The BOWMAN architecture was officially presented in March 2000 by Merugu et al. [32], as a joint venture between the University of Maryland, the University of Kentucky, and the Georgia Institute of Technology. The BOWMAN architecture represents a *node operating system*, on top of which composable active network elements can be built (a collection of such elements that form

an active network implementation is known as an Execution Environment).

BOWMAN is constructed on top of a standard host operating system, and provides abstractions of low-level operating system functions for execution environments through the NodeOS API. The current implementation is for System-V Unix, but a port for any POSIX-compliant hybrid of Unix is possible. Additionally, BOWMAN implements a subset of the emerging DARPA Node OS interface (see Section 2.7.3).

From BOWMAN's perspective, the users of the architecture are the execution environments that make requests via BOWMAN to the underlying operating system objects. Typical requests are in the form of transmission scheduling, CPU cycle reservation, global state updates and queries (such as route tables and shared memory), and storage requests. To clarify, execution environment is the term given for an active network implementation that calls upon the BOWMAN NodeOS to provide basic end-to-end services and/or provide some form of network programmability. Finally, *active applications* use the services of execution environments to send data with network service customisations. Hence, the role of BOWMAN is simply to provide a node operating system that safely supports any number of independent execution environments.

The main design goals for BOWMAN are: the support for per-flow processing with minimal overhead and fast packet processing, the provision of a *fast-path* to allow redundant processing to be bypassed, the provision of a global network architecture to support any number of virtual networks, and finally, the maintainance of reasonable performance for both packet forwarding and packet processing. Results from recent trials of the BOWMAN architecture show that 100 Mbps rates of throughput are obtainable for IP payloads of only 1,400 bytes. As can be seen from from Figure 2.8, this throughput rate is only slightly less than that of most user-level C packet forwarders. Additionally, the BOWMAN architecture runs exclusively in user mode, which not only shows that the system is efficient, but it also has the potential to provide a high level of safety.

The BOWMAN NodeOS makes three key abstractions, namely Channels, A-Flows, and State-Stores. As the name suggests, channels are simply links

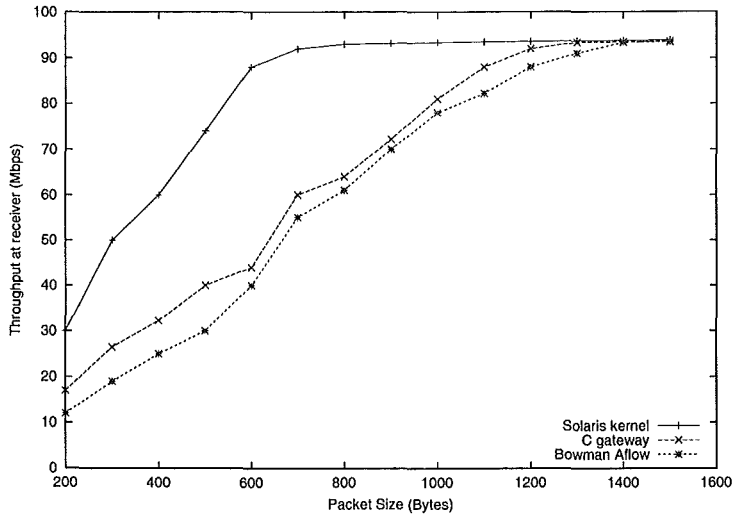

Figure 2.8: The rate of throughput for a single-hop data transfer for the BOWMAN architecture using various buffer sizes.

between endpoints in a BOWMAN virtual network, which can be created, modified, and deleted through the NodeOS API. Channels support a number of network and link-layer protocols. A-Flows represent independent points of computation within the network—each with their own execution context such as local variables, associated worker threads, and connected channels. Again, the NodeOS provides routines for the manipulation of A-Flows. Finally, State-Stores provide the information registry for all A-Flows on a given node, again with associated API routines.

The BOWMAN architecture also provides *BOWMAN Extensions*. These extensions are usually in the form of routing protocols, queueing mechanisms, and other network services. Such services are invoked via BOWMAN system calls, and are available via the system API.

The CANES Execution Environment

The CANES execution environment utilises the BOWMAN NodeOS API to allow network programmability on a per-flow basis. Active applications based upon the CANES execution environment include Active Error Recovery and Iterative gather-compute-scatter [31]. Results showed that only slight perfor-

mance degradation was experienced at each active node. Whilst little other information is available about these applications, the OS-savvy interface provided by the BOWMAN API suggests that many complex applications are possible, with an underlying active node operating system that provides unprecedented rates of throughput for any active network architecture.

2.3.8 SNAPd

The SNAPd system was introduced in April 2001 as part of a critical review of the current active packets paradigm [34]. As background information, the authors of the system claim that all active packet based networks do not offer a fully *practical* service, leading to the current inactivity of related research and development of active packet based architectures and applications.

Following the critical review, the authors introduced a *Practicality Framework* in order to assess the current packet based architectures in terms of safety, efficiency, and functionality. Subsequently, the SNAP packet language was developed to address the shortfalls of the previous systems (see Section 2.5.2). The SNAPd architecture was then introduced as the daemon process to load and execute SNAP active packets.

The SNAP packet language and the SNAPd host environment together form an architecture that addresses the issues of safety, efficiency, and functionality. With 53 active node primitive instructions which closely resemble bytecodes, the system is not as expressive as other systems, but this does work to SNAPd's advantage in terms of safety and efficiency. Additionally, the architecture runs entirely in user mode, yet still yields up to 76 Mbps throughput, as presented in Figure 2.9. However, as Figure 2.10 reveals, the latency at each hop is over 50% greater than the rate incurred when performing conventional IP routing.

The SNAPd environment is yet to be fully tested. The architecture has not yet been trialled with a realistically sized network, nor have any non-trivial applications been developed or tested to prove the systems practicality (listed by the SNAPd authors as a key issue for all previous architectures). However, it must be stated that the SNAPd architecture is in its infancy, and given the fact that it can provide security and solid performance rates in

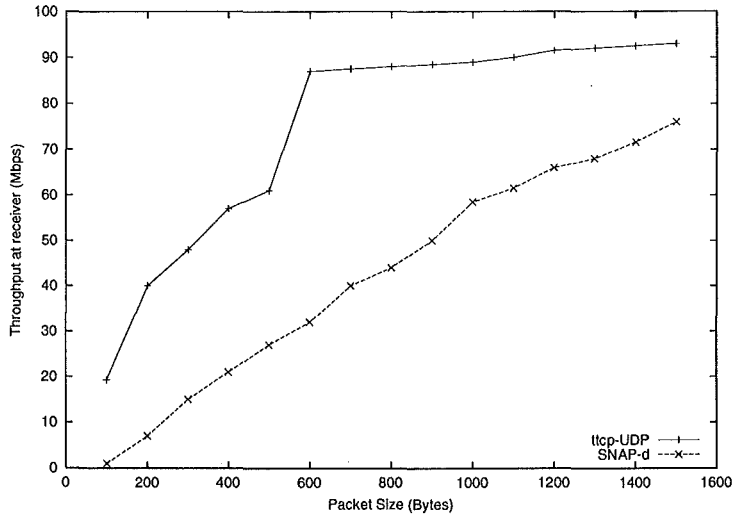

Figure 2.9: The rate of throughput for a single-hop data transfer for the SNAPd architecture using various buffer sizes.

user address-space, the case for active-packet based networks has been given new light.

2.4 Safety and Security Frameworks

By virtue of programmable networks, new risks are associated with the network endpoints, intermediate active nodes, and the very state of the network itself. Such risks reside in the form of unauthorised access to in-process or out-of-process data, and excessive memory, CPU cycle, and bandwidth consumption. *Safe* active networks protect known and trusted users and applications from breaching the specified bounds on such resources, where consumption is either malicious or in error. *Secure* active networks protect network resources from such over-consumption by untrusted users and applications.

Whilst a well-designed active network architecture can enforce restrictions on network resource consumption (or minimise the effects when limitations are breached), recent advances in active networks have seen the introduction of active network-specific safety and security frameworks to formally verify

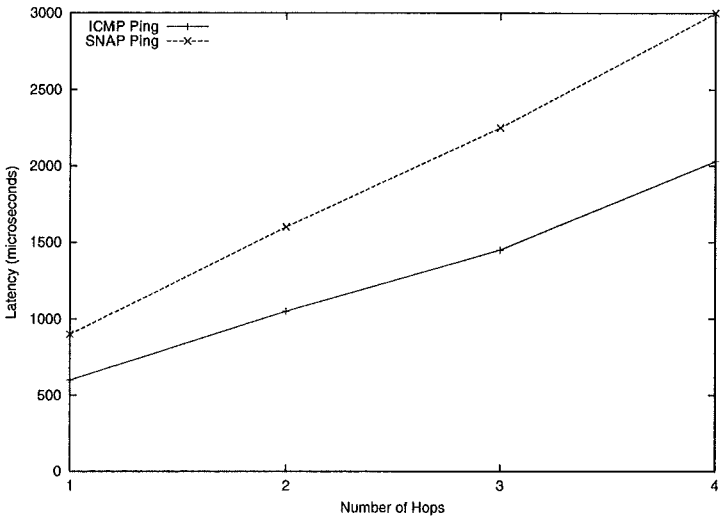

Figure 2.10: The latency of multiple-hop `ping` packets for the SNAPd architecture using a 1,500 byte buffer size.

that a system is safe, or to enforce security on an existing architecture. This section presents two such frameworks.

2.4.1 SANE

The SANE (Secure Active Network Environment) provides a layered architecture for the construction of secure active network infrastructures. Based on a minimal set of trust assumptions and rules, the framework securely bootstraps the remainder of the system, providing authentication and naming services for active packets. This provides a foundation proved as secure for execution environments to run in, with guaranteed integrity of services. The Switchware project uses this environment as the basis for all security services, ranging from verification of the initial network state through to the authentication of remote packets. Additionally, the ANTS architecture can utilise the SANE framework for all security services [3].

Static checks are used to verify the state of the system following startup, via a secure boot-loading architecture. After verification of the system's integrity, SANE performs dynamic checks on a per-user or per-packet basis (a public/private key scheme for the authentication of trust relationships

is utilised). Security is maintained through inter-node authentication, the restriction of the execution environment (and the programs the execution environment evaluates), and the provision of a partitioned naming service for users, nodes, and packets. By associating procedure names and objects with access privileges, authorisation is possible throughout a distributed network.

To maintain an efficient service, the SANE framework was designed to provide dynamic checks as quickly as possible, as they are performed most often. Static checks take considerably longer, but this is less important because such checks are normally performed once only, at system startup. Current research efforts of the SANE framework focus on developing optimal tradeoffs between costly static checks and inexpensive dynamic checks. Potentially, an expensive compile-time or boot-time static check could eliminate the need for current dynamic verification on a per-packet basis.

The cost of authentication via the SANE framework appears to be acceptable (albeit there are no other frameworks to use as a comparison). By sending active ping packets, an authenticated ping took an average of $8\mu s$, whilst an unauthenticated packet took $5\mu s$. Data packets performed worse, with a 28% degradation in performance for received packets, and a 62% degradation for sent packets.

The SANE framework is well-suited to execution environments that do not provide their own security services, for a moderate degradation of performance. Alternatively, the reliance on a safe packet language (see Section 2.5) may provide adequate safety. This is displayed by the PlanET architecture, where the PLAN packet language provides safety for program packets that do not require special privileges [23].

2.4.2 ActiveSpec

ActiveSpec is a framework for the formal verification of security policies for active networks and their services [16], guaranteeing the integrity of an active network architecture. Through a formally defined set of services, policies, and resources, the interactions within an active network can be modelled with subsequent verification of security requirements. A framework is provided that allows the individual components of active networks to be specified under

various representations. Once the specifications have been defined, the PVS theorem prover is used to formally verify the properties under question [25].

The ActiveSpec framework requests a simple formal specification of active nodes (and their resources) and security policies, and verifies through PVS that packets have access to secure resources only if they present the correct permissions (the preservation of network resources such as bandwidth consumption has not yet been addressed). This is achieved by monitoring the incoming and outgoing channels for any given packet, as well as monitoring the resources that the packet consumes (the state of the related active nodes is also checked after processing the packet). If any violation of resources occurs, the verification process returns an error.

The ActiveSpec framework was used to model and formally verify the integrity of the boot-loading service for the SANE environment, guaranteeing a secure boot process for active network architectures that do not implement their own security mechanisms. Current research efforts for the ActiveSpec framework are concentrating on the extension of the system to verify packet delivery and packet evaluation primitives for generic architectures.

2.5 Packet Languages

As made apparent during the implementation of architectures such as the SwitchWare project and SNAPd, languages specific to active networks could prove important in addressing key issues such as safety, portability, flexibility, and efficiency. Such languages attempt to make the execution of arbitrary user instructions on active nodes safer by restricting the dangerous constructs of conventional languages, and by placing bounds on the resources that a program may request.

Whilst architectures such as ANTS attempt to provide holistic environments to cater for all demands that active applications place on their host networks, the introduction of active network-specific languages permits a looser coupling between the underlying network architecture and the network interface that the active application calls upon. This section presents several such active network-specific languages.

2.5.1 PLAN

PLAN (Packet Language for Active Networks) is a lightweight language which acts as a replacement to packet headers in a conventional network (packet headers themselves can be viewed as very restricted programs) [22]. PLAN programs are very restricted in functionality (only a handful of primitives exist), but they are also permitted to call a second level of node-resident executable code known as *service routines*. Such service routines are normally written in more powerful native languages to provide core network functions.

The language itself is based on lambda calculus, with a very simple grammar and only twelve data-types, five primitives, and the ability to define and call PLAN-based functions (which can optionally raise exceptions). Primitives such as `OnRemote` and `OnNeighbor` allow service routines to be run at either the next hop in the active network or at the specified end-point. After a host application constructs a PLAN packet, it is injected into the network via a port on which the PLAN interpreter is listening. All communication between the active network and the host application is performed via this port.

To address safety issues, any program written in PLAN is guaranteed to observe the user-defined bounds on resources, in particular network bandwidth, memory, and CPU cycle usage. Additionally, the language is type- and pointer-safe, providing strong static guarantees that the PLAN interpreter can not be manipulated to perform arbitrary operations. Additionally, concurrently executing programs cannot interfere with each other because the PLAN language is stateless.

Security issues have not yet been addressed, but the restrictive nature of the PLAN language implies that compromises in security will not compromise the entire state of the network. According to the authors, it is envisaged that security services from active network host environments will provide protection and authentication where necessary, such as that offered by the SANE environment (see Section 2.4.1).

The design principle of the language is to include only core functionality—features of the language that are considered necessary and that also impose no safety risks. Subsequently, constructs such as recursion and unbounded

iteration have been removed. This, coupled with an automatically decrementing resource counter, suggests that all PLAN programs terminate, albeit the programs are restricted in expressibility and functionality.

As discussed in Section 2.3.4, the PlanET active network was trialled using the SANE host environment and the PLAN packet language, with throughput rates reaching 48 Mbps on a 100 Mbps Ethernet network. Considering that this network was built from the ground up (no underlying IP layer was used), PLAN promises to be a safe and flexible language in which to build active applications.

2.5.2 SNAP

The authors of the SNAP packet language (Safe Networking with Active Packets) claim that SNAP is the first language to be both safe *and* efficient [33]. Whilst PLAN is a safe packet language, the authors of SNAP argue that performance evaluation of PLAN reveals that PLAN is too inefficient to be utilised at every active node in end-to-end communication. Hence, SNAP was developed to provide both safety and efficiency, with initial results indicating that these goals have been achieved (up to 80 Mbps throughput can be achieved in the SNAPd host environment, executing SNAP program packets in user mode).

The scheme of the language provides a simple set of primitives that are then compiled to byte-code, ready for injection into the active network. The limited expressibility of the language has allowed the authors to assert safety theorems about all SNAP programs, guaranteeing the protection of network resources, in particular the safety of CPU, memory, and bandwidth usage, as well as guaranteeing the isolation of separate processes. Additionally, this limited expressibility allows a high level of efficiency to be achieved, with no substantial degradation of performance at each network hop.

Similar to the PLAN language, SNAP programs are guaranteed to terminate, but with the enhancement that they also run in time that is linear to the program length. Additionally, particular attention has been paid to developing the SNAP language in order to permit a fast interpretation by active nodes. One of the most notable features of the language is that all

stack values are located at 32 bit offsets, reducing byte-alignment overheads during interpretation.

As mentioned previously, the SNAPd interpreter follows the formal semantics of the SNAP language to safely interpret program packets. The SNAPd interpreter runs entirely in user mode, utilising UDP for packet transmission. After implementing a SNAP version of the ICMP `ping` program, the language was tested on a five hop, six node IP-based network, using the standard Linux kernel router as a comparison.

Of the more notable results, the SNAP latencies were very similar to those of the benchmark latencies. Whilst the kernel mode benchmark could saturate a 100 Mbps network with a 500 byte packet size, SNAP obtained an 80 Mbps throughput rate once the packet size reached 1,500 bytes. Not surprisingly, profiling revealed that the copying of data from user address space into kernel address space accounted for the majority of the overhead, suggesting that a kernel mode SNAP interpreter will provide performance similar to that of the Linux kernel router.

As a final comment on the issue of functionality, the authors of the SNAP packet language are currently working towards a PLAN-to-SNAP compiler. This will enable a new level of operability between active applications and active networks, as well as presenting several new research possibilities.

2.5.3 SafeTCL

Whilst the SafeTCL scripting language [37] has not yet been utilised by any of the surveyed architectures, the language did provide a precedent for a secure language in an untrusted public environment such as the Internet. The designers of SafeTCL determined a set of unsafe features in the language, and then disabled them from being interpreted. Additionally, any features of the language that had the potential to consume resources (such as the `puts` command in which disk space or memory can be written to) could now be overridden within a program to impose user-specified bounds.

SafeTCL implements a simple *padded cell* model to protect the underlying operating system from unauthorised access and resource consumption, much in the same way as a modern operating system protects its own user processes.

This, coupled with a restrictive core set of primitives, provides a safe scripting language to embed into applets and other web-based applications.

To address security, data and code are semantically grouped together, resulting in an overall reduction in the amount of security-aware code required. This design principle was motivated by the recent merging of data and code in Internet based services such as email. Additionally, SafeTCL modularises its security policies to decouple security constraints from the associated code. This results in unrestricted access to security policies for analysis and design purposes, and subsequently promotes the reuse and composition of existing policies.

One very important characteristic of the SafeTCL language is that it can be compiled into native machine code, and/or call or be called from C, resulting in high performance code modules. This is important, as a future active network architecture could compile SafeTCL source-code at run-time on demand—maintaining machine independence yet offering an expressive, secure, and efficient active network.

2.6 Host Implementation Languages

Previous studies of active networks have shown that the selected implementation language can have a substantial effect on the performance, functionality, and security of the given architecture [35, 52, 33, 22, 21, 26]. All of the high-performance architectures surveyed in this chapter implemented their packet interpreters (otherwise referred to as execution environments) in the languages of C and C++, whilst the remainder of the architectures used Java. This section discusses the characteristics of the above-mentioned languages, followed by a brief introduction to a new programming paradigm for active networks—that of middleware and distributed systems programming.

2.6.1 C and C++

If performance of an active network architecture is a key issue, then it is likely that the implementation language will be C. C++ offers reuse benefits in the form of inheritance, design benefits in the form of improved modu-

larity, and autonomy in the form of polymorphism, but these benefits come at a cost—applications that utilise such features of C++ usually incur a performance penalty with a range of 10%–25% when compared with a pure C implementation.

Embedded C and C++ compilers exist for most architecture and operating system combinations; subsequently most routing devices utilise a subset of either language, albeit a proprietary subset for most vendors. Additionally, most operating systems allow user-defined applications to access system calls via C interfaces, which enable active networks to perform low-level operations such as direct communication with network interfaces, direct memory access, and inter-process communication.

Security is not provided by C or C++; the languages are not type-safe, meaning that one type can be substituted for another in various ways, either to provide flexibility or simply in error. Arbitrary segments in memory can be accessed, and depending on the operating system, results could be catastrophic. Obviously, additional safety checks must be imposed if an active network is to accept C program packets. Finally, development time using C and C++ is usually much longer than other high-level, network-centric languages, such as Java or TCL.

2.6.2 Java

The main barrier to Java's success as the leading implementation language for active network architectures is that of performance. Even under fast hardware (dual Intel Pentium IIIs, 512 Mb primary memory), applications written in Java are noticeably slower than natively-compiled languages, suggesting that the price paid in performance is constant due to run-time overheads, regardless of the hardware specification. Whilst just-in-time and ahead-of-time Java compilers exist [48], currently only a 5%–20% increase in performance is possible. The poor performance of Java in terms of latency is evident in the ANTS active network implementation [52].

Performance appears to be the only shortcoming of Java. Java is network-centric, meaning that network functionality that had to be hand-coded using C is standard with any release of Java. Additionally, using Java's Remote

Method Invocation interface (RMI), classes can be written and bytecode-compiled by network users, inserted into active packets, and then interpreted by any active node that has access to a Java interpreter. Furthermore, using Java's *Reflection* API, arbitrary classes can be received and inspected by active nodes, requiring only minimal constraints on the format of active packets. The technique of object reflection forms the basis of a related research project by Curran and Parr [15], where arbitrary classes are loaded and executed to perform QoS routines for a multimedia server.

As noted by Krupxzak et al. [26], Java is well-suited to the deployment of new active network protocols because of its network, run-time type information, and portability features. Additionally, safety is another of Java's favourable characteristics. Not only is Java relatively type-safe, the elimination of raw memory pointers from the language means that arbitrary memory cannot be accessed directly, making it increasingly difficult to compromise the state of the running program and/or the underlying interpreter. The experienced Java programmer can still exploit anomalies in the Java virtual machine specification to compromise system safety (anomalies such as optimisations made to visibility checking), but Java still offers a vast safety improvement over compiled languages such as C.

2.6.3 Distributed Systems and Middleware

Whilst Chapter 4 provides a detailed overview of middleware frameworks and distributed systems, it is worthwhile noting the importance of such platforms when discussing active network implementation languages. Several recent active networks, programmable networks, and distributed QoS frameworks have utilised middleware frameworks (such as CORBA) for at least part of their system's implementation [4, 17, 42, 28].

Middleware frameworks such as COM and CORBA offer inherent support for the registration, loading, execution, and control of user-defined classes—attributes that most active network architectures attempt to implement internally. Additionally, both CORBA and extended COM (referred to as DCOM) offer enhanced distributed object services, such as remote method invocation and inter-host communication via distributed event handling. Java's remote

method invocation API also provides solid distributed application support.

Both the COM and CORBA communities have also endeavoured to make their middleware environments as secure as possible, due to the potentially dangerous task of running remote processes. Accordingly, both frameworks offer a rich set of routines to govern the conditions in which remote processes can be started, accessed, and terminated. Additionally, levels of resource usage can usually be monitored, if not restricted, with a fine level of granularity over the interaction with the remote host running the process. Finally, both frameworks offer support for the marshalling of arbitrary data, including the serialisation and distribution of objects between several hosts, which is well-suited to the demands of active network infrastructures.

Given the close match in terms of services between those that middleware frameworks offer and those which active network implementations require, it is surprising that no previous active network architectures have fully utilised the services of middleware frameworks. Consequently, Chapter 4 presents such an architecture.

2.7 Specialised Protocols and Interfaces

The previous sections in this chapter have described the typical active network architectures and the various components within, yet the aspect of connectivity within and between active networks has not been addressed. Hence, this section presents the draft protocols and interfaces proposed by the active network research community to provide a standard for both the inter- and intra-operability of active network architectures.

It should be noted that all the protocols and interfaces presented in this section are still draft documents, with considerable current research attempting to identify the core set of services and routines required to finalise these proposals as official standards.

2.7.1 Programmable Network Interfaces

In the context of active networks, it is worth noting the IEEE P1520 standards initiative for programmable network interfaces [7]. The IEEE P1520

working group introduced a draft standard for a networking application programming interface in January 1999, which, in part, is intended to assist the facilitation of active network policies via various P1520 interfaces and options. Not only active packets can be transmitted via a P1520 interface; for example, video transcodings could be specified within a class of a multicast tree, supporting hosts with varying transmission and decoding rates.

As an overview of the P1520 draft standard, it is envisaged that all hardware devices will support a minimal set of interfaces at several different layers to provide open systems communication, primarily for ATM and IP routers (see Figure 2.11), as well as for switches and bridges. The objective is to develop an open system for signalling and device management/control, as well as higher level multimedia services on networks. By leveraging a distributed object oriented approach, third-party service providers can rapidly integrate new protocols, and hardware vendors will be decoupled from the software requirements.

The importance of the P1520 standard for active networks is that a well-known set of interfaces will be provided, allowing communication with execution environments, which could become a global hardware standard upon finalisation. However, in some ways the standard could also challenge active networks as the new paradigm for providing programmable networks, as the standard proposes a layered architecture very much in parallel with that of current active network architectures, ranging from the physical elements through to high-level APIs, to provide personalised end-user applications and services. A more likely scenario would be the adaptation of the P1520 interfaces into proprietary hardware, with the implementation of active network services through well-known P1520 interfaces.

2.7.2 The Active Network Encapsulation Protocol

All execution environments within active networks must determine and classify active packets, yet the issue of *how* such classification is performed has, up until recently, been implemented in an arbitrary manner by the various active network architectures. In July 1997, the Active Network Encapsulation Protocol (ANEP) was introduced as a draft RFC, to encourage a standard-

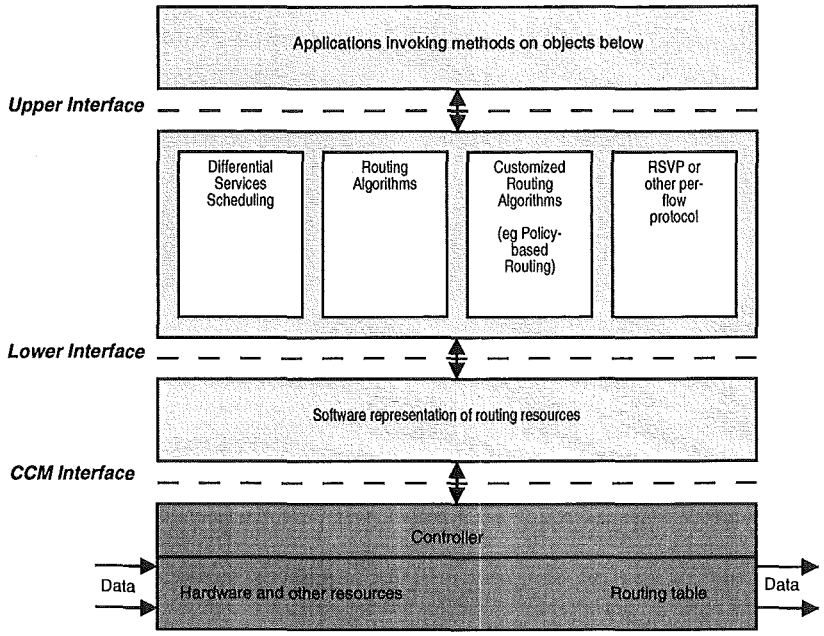

Figure 2.11: Mapping of the P1520 architecture to IP routers/switches [7].

ised active packet frame format [1].

The ANEP protocol was primarily proposed to provide a formal mechanism for the encapsulation and transmission of active network frames. Encapsulated frames can be transmitted over network infrastructures such as IPv4 or IPv6, or transmitted directly over the link layer, as presented in Figure 2.12. Additionally, the ANEP format is as general as possible to allow the co-existence of different execution environments, promoting both extensibility and ongoing active packet research. A simplified format also allows an efficient demultiplexing of received packets, and provides existing network-layer routers with a straightforward adaptation of the ANEP protocol.

An additional benefit from such an encapsulation protocol is that devices that support ANEP will only be subjected to minimal default processing when a requested execution environment is not available. Also, information that does not fit conventionally into active packets (such as active network security messages) can be processed by ANEP-aware devices, much in the same way that ICMP messages are encapsulated within IP packets.

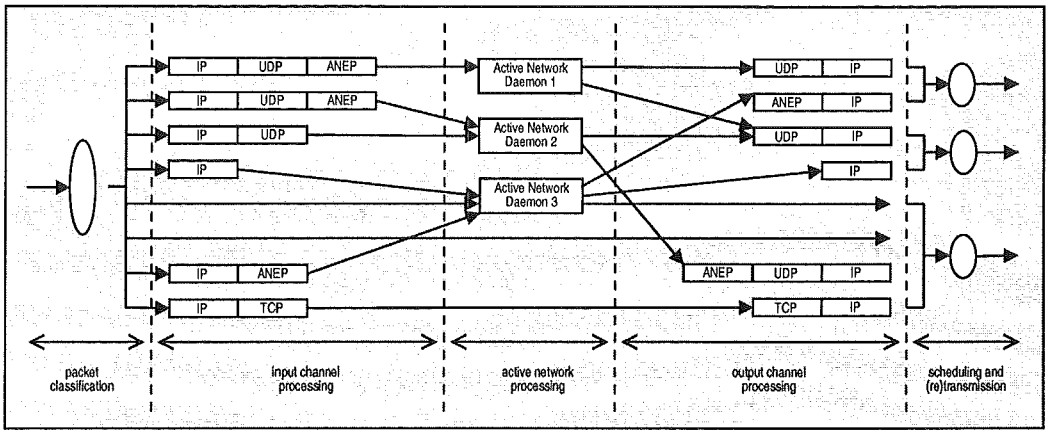

Figure 2.12: Possible ANEP-encapsulated packet flows through an active node [12].

ANEP-compliant execution environments will place data from an active application and/or code modules into the payload of one or more ANEP frames. Options in the ANEP header include authentication, confidentiality, and integrity information. Type identifier fields are initialised with the identification of the relevant execution environment. It is the responsibility of the active network architecture to deliver the packet to the appropriate active node, but because the ANEP protocol is encapsulated, IP is normally used to deliver ANEP active packets.

It should be noted that execution environments that support ANEP packets can also process non-active packets. For example, if a non-active application sends a legacy packet, the execution environment could proxy the packets via newly formed ANEP packets, much in the way split-media routers transport streams between network borders using differing protocols. Alternatively, the execution environment could establish a private channel that uses a legacy network protocol to provide default forwarding services for the encapsulated packets.

2.7.3 The Node OS Interface

Following the IETF Active Network Working Group's first version of the architectural framework for active networks [11], the *NodeOS Interface Specification* for active node operating systems was proposed to establish well-known core active node services [39]. Benefits from such an interface standard include the ability for a single node to support multiple languages due to the separation of the operating system and the execution environment, and the ability to rapidly port a node operating system to new hardware types.

The NodeOS Interface Specification defines a basic set of functions to access and manage abstractions of the resources associated with an active node, including communication, memory, and computational resources. Hence, the NodeOS acts as a layer between the execution environments and the underlying physical resources. The primary role of the interface is to support packet forwarding, with a secondary role of executing active packets (this is primarily the role of the execution environments that call the NodeOS interface). The NodeOS interface is based upon the concept of packet flows, which implies that packet processing, admission control, and accounting is performed on a per-flow basis.

It is not assumed that all NodeOS implementations will support the same set of routines. Although all implementations must support the core set of routines, many NodeOSs will provide additional interfaces and routines that specialised execution environments will call upon. The main requirement for a NodeOS implementation is that packets are forwarded as rapidly as possible—in hardware if packets are non-active. Finally, whenever a NodeOS implementation requires a service that is not governed by existing active network specifications, the NodeOS should refer to established conventions such as POSIX.

The NodeOS specification also incorporates security. Requests made to a node operating system on behalf of a particular user, application, or service must be accompanied by credentials sufficient to verify that they originated from an entity authorised by the active node and/or execution environment. However, like most interface specifications, no direction is given regarding *how* such authentication is to be performed—this is left to the implementors.

As mentioned in Section 2.3.7, the BOWMAN node operating system implements the current draft set of NodeOS routines, with throughput rates that come close to saturating a 100 Mbps Ethernet network, and with only slight performance degradation at each additional network hop. Whilst this is the only known implementation of the NodeOS interface, the favourable results, coupled with the benefits of an active network-aware operating system, strengthen the case for a standardised active network foundation.

2.7.4 Protocol Boosters

Protocol boosters are robust protocol adaptors designed to dynamically improve the performance of protocols in heterogeneous distributed computing systems, on an end-to-end basis [29]. A protocol booster compliments an existing protocol by adding, deleting, delaying, or modifying selected protocol messages and/or payloads, but a protocol booster is not permitted to replace or terminate the existing protocol. The motivation for the development of protocol boosters is as follows: many protocols involve inefficiencies when they are applied as general purpose solutions to the delivery of application and network requirements, and protocol boosters fine tune such protocols to a specific environment.

The traditional approach to protocol design methodology is to cater for the worst case scenario and suffer inefficiencies during normal protocol use. With the introduction of protocol boosters, protocols may be designed for average-case or best-case scenarios, with the dynamic loading of protocol adaptors to accommodate for worst-case scenarios—when they occur. Additionally, such protocol adaptation allows for the optimisation of specific applications or network conditions. An example of the successful use of protocol boosters was in the form of a Forward Error Correction (FEC) protocol booster, used on a 155 Mbps ATM link running TCP/IP. When a Bit Error Rate (BER) of 10^{-4} or greater was introduced to the link, TCP completely stalled, but managed to operate normally with the FEC protocol booster.

The application of protocol boosters on demand may be considered as a structured subset of active networking, given the presence of a programmable network infrastructure. Such protocol boosters could be easily deployed by

active networks, with no need to update the underlying infrastructure.

2.8 Active Network Backbones

Because of the increasing number of active network architectures and components, the Active Network Working Group established a DARPA-funded testbed referred to as the ABone [5]. This infrastructure is intended to support the melding of related research projects, as well as to serve as a large virtual network for the deployment and testing of new protocols and services.

The ABone is designed to be a sharable resource, open to all active network researchers and environments. It utilises existing network links, including the DARPA-funded CAIRN testbed. Internet overlays are used to link active nodes from remote sites (see Section 2.8.1). The ABone is intended to have high availability, with little or no pre-arrangement of service setup.

Multiple node operating systems are also supported, although all are Unix-based thus far. The ABone provides a considerably large infrastructure, with the ability to support 1,000 active nodes at present. Security of the ABone is currently the responsibility of the local node administrators, who must ensure that their own site's security, and that of all others, can not be compromised. Finally, some central coordination is required to organise the deployment of new software and the ongoing maintainance of the network state. This is provided by a small organisation named the ABone Coordination Center (ABOCC), who are also DARPA funded.

The basic components of the ABone consist of the permanent execution environments, the core active nodes and edge active nodes, and a naming and registration service maintained by the ABOCC. Additionally, the ANetD service exists on each active client node in the ABone network, which is used to install new execution environments, and performs global housekeeping functions such as event polling, packet demultiplexing for nodes that support multiple execution environments, and node management.

Future work with the ABone will see the establishment of extensions to ANetD in order to close known security holes and implement stronger authentication via digital signatures. The possibility of caching executable code is also being examined at present. The conversion of ANetD to a kernel-level

process is also being considered, to improve performance and support dynamically loadable kernel modules such as network device drivers. Finally, pushing ANetD down to the link layer will allow direct communication between ANEP and the network devices, improving performance by bypassing the IP stack altogether.

2.8.1 Internet Overlays

Whilst the ABone infrastructure works transparently within the confines of the main DARPA testbed, some form of bridging mechanism is required to allow a distributed testbed for global convergence of active network architectures and components. Accordingly, the Active Network Overlay Protocol (ANON) was introduced as a draft RFC in December 1997 [8].

ANON allows the interconnection of ABone-based execution environments, in both point-to-point and bus topologies. The protocol is dynamic, hence execution environments and nodes can subscribe and sign-off at any time. ANON does not pre-empt existing or proposed active network routing or naming protocols, rather it simply supports a foundation to house such protocols.

The ANON overlay performs the interconnection of remote active nodes via the ANEP encapsulation protocol. ANON network elements exchange packets via the ANEP addressing mechanisms, but ANON network elements do not forward ANON packets. Hence, if an incorrectly addressed packet arrives at an ANON-based remote site, it is silently discarded, as the underlying ANEP mechanism is responsible for frame routing. Internal ANON routers are permitted to dispatch and relay packets with an execution environment, delivering the ANEP payload to the recipient in the form of an active packet.

Chapter III

Unresolved Issues of Active Networks

There are several unresolved issues of active networks—namely efficiency, functionality, and security. Unfortunately, these issues are also the key goals for active network research, indicating that new directions in the implementation of active networks may be required. Whilst many architectures can resolve two of the three key goals simultaneously, the remaining third goal tends to be compromised, limiting the total practicality of the given architecture.

3.1 Efficiency

Efficiency is the key area where active networks fail to achieve practicality. Although only limited results have been published by active network researchers, it appears as if a marked deterioration in performance is likely when several network hops are made between the source and destination of a stream [24, 33].

3.1.1 Performance Ceilings

Due to the lack of research into throughput rates for multiple hop active networks, it is difficult to characterise the general performance bounds common to active network architectures. However, from experiments performed on the PlanET architecture, it is apparent that a ceiling is imposed on the performance of the system after the first hop, regardless of implementation specifics (such as type of execution environment) [24].

By comparative results, this performance ceiling does not exist on passive networks. For example, after the second hop was visited on a 100 Mbps

Ethernet WAN, all PlanET active node traffic throughput rates had been limited to a maximum of 60 Mbps, whilst the non-active links maintained 95 Mbps throughput. A likely explanation for this behaviour is the impact of data copying that must be performed by all active routers. From the second hop onwards, active routers must copy the data from the incoming interface, process the data, and then copy it to the outgoing interface. If each node in the test network has identical hardware specifications, it is likely that no further degradation in performance will be experienced on the downstream active nodes after the initial performance hit.

3.1.2 Performance Degradation

It is expected that a level of degradation is to be incurred at each network hop because of the additional processing required, but the context switching from kernel to user mode, combined with multiple data copies per packet, means that most architectures can only provide a practical transport mechanism for small-scale networks. In addition, no results have been published where the effects of the number of hops on application data transfer have been measured—the status quo suggests that `ping` is a suitable application to infer an entire network's characteristics. Whilst `ping` is useful for testing latency, a more realistically sized benchmark such as file transfer is also required to provide insight into a network's performance under varying conditions.

3.1.3 Packet Classification Time

Only the BOWMAN architecture has reported results from the analysis of applying multiple code modules per packet stream [32]. The results indicate that BOWMAN's packet classification system (the mechanism used to invoke code modules on candidate packets) is capable of classifying 1.3×10^5 packets per second with up to 256 code modules per packet. This is equivalent to a throughput of 1.5 Gbs, where typical 1,514 byte IP packets are used. Using BOWMAN as the only example, it could be assumed that the task of relating packets to requested code modules does not have a significant effect on performance of active networks. However, many more systems need to be trialled before this assumption can be asserted with any degree of confidence.

3.1.4 Packet Processing Time

The amount of time an active node requires to process a packet according to its associated code modules has not yet been addressed by active network research. A weak inference could be made by looking at the throughput rates for various systems, but little mention is made of the actual *active* processing performed at each hop, or the number of hops associated with the throughput results. A suite of benchmarking routines would be beneficial to the objective analysis of packet processing performance, such as a compression routine, a sorting routine, and an encryption routine.

3.1.5 Processing Optimisations

As emphasised by the development of node operating systems, packet processing optimisations, by way of resource scheduling and alternative routing, can introduce considerable savings in terms of processing time [39, 32]. For example, the BOWMAN architecture provides a *fast-path* where special channels are established for packets that can be predetermined to require no additional processing, hence avoiding the costs of data copying, context switching, and multi-threaded processing. This technique is conceptually similar to that of *IP Switching*, where only the first packet in a stream is routed, and all following packets are labelled and then switched at wire speed.

3.2 Functionality

The functionality of active networks is difficult to quantify objectively. Whilst most active networks offer the same degree of processing options via their packet languages, the expressibility of the language determines the ease with which active applications can be developed. Additionally, functionality can be found in the level of control over packets and active routers in the network, as well as in the amount of effort required to deploy new code modules for node-based architectures. Usability, in terms of a simple yet powerful API for users and active applications, also defines the level of functionality within an architecture.

Unfortunately, very few active networks give details on the above-mentioned

architectural features. Of notable findings, the ANTS architecture did provide a code-module deployment scheme, the PAN architecture gave full details of its user API, and both the PLAN and SNAP architectures gave the formal grammars of their packet languages. However, no architecture thus far has presented all details in terms of functionality.

3.2.1 Stream Granularity

No architecture has discussed the level of granularity when defining the invocation of code-modules on packets. For example, whilst most systems specify that a stream of data can be processed by a code module, no mention is made as to whether processing is performed on the outgoing data, the incoming data, or all packets in the stream. Additionally, the ability to enforce *mandatory* modules on all packets that are routed through an active node has not been discussed as of yet. On the other extreme of granularity, no system has discussed the possibility of specifying the invocation of code-modules on a packet-by-packet basis for node-based architectures.

3.2.2 Packet Language Expressiveness

The expressiveness of a packet language is the key characteristic in determining the functionality of an active network architecture. The easier it is to access resources such as bandwidth, memory, and processors, the more rapidly active applications can be developed. Whilst less expressive languages tend to enhance security, restrictions are placed on the bounds of resultant code-modules, unless experienced programmers devise low-level workarounds. The expressiveness of a language also has a strong correlation with code-module development times, using the migration from assembler to C as an example.

3.2.3 Code Module Deployment

The ability for application programmers to easily deploy code-modules is another indicator of the flexibility of a given active network environment. Unfortunately, only the ANTS architecture has provided details on code-module deployment mechanisms. Disregarding the lack of research into such

deployment mechanisms, a system that allows extensive code-modules to be created for dynamic protocols, yet requires system down-time for such modules to be deployed, surely can not be considered truly flexible.

3.2.4 Usability

Usability can be determined by the ease with which an active application can access and utilise the services of an active network architecture. For example, a complicated architecture with numerous conceptual components is not as usable as a simple architecture that provides the same services. Accordingly, a cohesive and simple API is considered more usable than one that offers many routines to perform the same task, with little consistency. The degree of run-time support to assist run-time debugging must also be considered when assessing the usability of an architecture [33].

3.3 Safety and Security

Ideally, the degree of safety offered by active networks must be at least equal to that of conventional IP [33]. However, it is very difficult to fulfil this claim, due to the increase in low-level programmatic control offered to end-users. As discussed in Section 2.5, considerable effort has been placed in protecting both the user and the underlying execution environment from programmatic errors or malicious use. However, the paradigm shift from a configurable network to one that is programmable inevitably leads to compromises in safety. The remainder of this section discusses such compromises.

3.3.1 Execution Privileges

The instinctive means of addressing poor active network performance is to execute the architecture in kernel mode, eliminating context switches and double-handling of data. In fact, this is exactly what the PAN architecture does [35]. Unfortunately, safety and security are compromised under this regime, as most operating systems perform no address-space checks to verify that the requested resources are legally accessible by the user (as the user is in fact the operating system under kernel mode execution). Unless the active

network architecture can provide a verifiably safe language and execution environment in which to run user-defined code-modules in kernel mode, kernel mode execution of code-modules must be avoided.

3.3.2 Resource Consumption

Even the cornerstone of networking, IP, is not immune from excessive resource consumption. If the *Time to Live* header option is ignored by IP routers, an entire network can be flooded within seconds. Similarly, *router convergence* caused by spontaneous updates of OSPF route tables can also cripple a wide area network. However, active networks are exposed to more severe risks than the above problems, due to their highly programmable nature.

Active networks are susceptible to excessive resource consumption in the form of memory, processor time, and bandwidth, as initiated by user-defined code modules. Particular attention must not only be paid to emulating the stability of IP, but also to ensuring that excessive resource consumption on active nodes and network links does not occur. The ANTS architecture, among others, implements *watchdog threads* to inspect the resource consumption for an active packet, with action taken to terminate the packet's execution if limits are breached. This is somewhat of a brute-force method to ensuring safety, and can most likely be compromised if the watchdog thread is attacked. Subsequently, other architectures rely on their restricted languages to block excessive memory consumption, but the experienced 'hacker' usually can find loopholes to such restrictions. In this light, node operating systems appear to be the most favourable to developers of active network architectures, as the operating system itself can concentrate on resource allocation, based on pre-established policies.

3.3.3 Memory Protection

It is worthwhile at this stage to note the importance of memory protection. Whilst most modern operating systems found in both workstations and routers protect out-of-process memory access (with the exception of shared memory), it is the responsibility of the implementation language to protect

in-process memory usage (refer to Section 2.5 for a detailed discussion). For example, Java provides no mechanism to access arbitrary segments of memory, so no active packet implemented on a Java-based active node can inspect another active packet's data without permission. Conversely, the C language permits such access—which can only be addressed by providing a restricted alternative version of the language, as present in the SmartPackets architecture.

Chapter IV

COMAN: Motivation, Design, and Implementation

This chapter introduces the COMAN active network architecture. The motivation for the development of the middleware-based architecture is presented, followed by a general overview of middleware technology. Next, the design of the architecture is outlined, including system configuration specifics and details of the core COMAN application programmer interface. System implementation details and a source code listing for a sample client application are also included.

4.1 Motivation

COMAN was initially implemented to prove the case for a functional active network architecture that could be rapidly developed via middleware. By using middleware services such as remote method invocation and in-built data marshalling, key aspects of the architecture's functionality could be rapidly developed, without the need for duplicating existing code. By reusing existing services, the architecture could also remain relatively simple and unconstrained, as opposed to the more conventional active network architectures.

4.2 Middleware Overview

With the continued growth in network-based applications, distributed middleware systems (or object request brokers) of various incarnations have emerged to manage issues of complexity and scalability. Common middleware systems include CORBA [20] and DCOM [14], with both systems available on Unix and Windows platforms. Introduced by the Object Management

Group, CORBA is a well-specified distributed object protocol that is implemented by various independent vendors, whereas Microsoft's DCOM represents a single entity consisting of a wire-protocol and a proprietary implementation. Java's Remote Method Invocation API is also gaining popularity as a web-based middleware system, with its system independence eliminating the requirement for platform specific implementations.

The theme of managed distributed object creation, remote method invocation, object persistence, and object termination is common to all distributed middleware systems. Accordingly, the application of middleware is suitable wherever complex distributed functionality is required. Middleware-based systems are commonly utilised when using distributed processing to increase performance, or when implementing redundant links to increase system reliability. From a design perspective, advantages of middleware systems include the separation of interface and implementation [18], support for objects with multiple interfaces, and location transparency. In the case of DCOM, language neutrality is also supported natively.

4.2.1 Existing Middleware-Based Distributed Frameworks

Several middleware-based frameworks have been developed to increase the level of support for next-generation distributed applications. Whilst distributed middleware systems are well-suited to conventional client/server based applications, enhanced frameworks have been introduced to provide middleware-specific performance optimizations and QoS features [45]. One such framework, TAO, is a real-time object request broker which provides high-performance middleware components for common networking tasks, with multiple-layer QoS support [27]. TAO has recently been used to implement a framework which supports dynamically-specified transport protocols to support low latency, high bandwidth data streaming. Through the flexibility of its middleware composition, optimised scheduling and buffering techniques were utilised to provide low latency, high throughput video/audio links, suggesting that middleware is a suitable mechanism to enable next generation distributed applications.

A similar framework for middleware-based QoS support has been pro-

posed by Becker et al. [4], which includes a Java-based prototype implemented within a gigabyte router testbed. Another Java-based distributed architecture, RWANDA, also focuses on middleware support for enhanced QoS services, in the domain of high-latency environments such as the Internet [38]. As the goal of the architecture is to provide a highly-adaptable service, a reconfigurable protocol stack to support arbitrary user-defined protocols has been implemented. The architecture also provides application-level control over common transmission and synchronization tasks.

4.3 Merging Middleware and Active Network Concepts

4.3.1 Distributed Middleware Systems

A close relationship exists between distributed middleware systems and active networks. Active networks aim to provide efficient, complete, and highly flexible network infrastructures, and distributed middleware systems appear capable of enabling such systems, either in part or in whole. Despite this close relationship, the mainstream approach to active network design appears to involve the internal implementation of mechanisms for distributed communication, library loading, remote method invocation, and security management, with little consideration for existing frameworks that support such tasks. The architecture presented in this thesis presents a case for active networks that are based entirely in middleware, avoiding the need to 'reinvent the wheel' at both the architectural and implementation levels.

Although active network frameworks provide a great degree of application-level flexibility, for the most part they are closed systems that do not provide services for existing legacy client/server applications. By utilizing middleware support, existing distributed applications may be easily modified to connect to active network architectures. In a similar manner, the presence of a well-defined middleware interface allows the bridging of different active network architectures, forming potentially global active networks without the necessity for an intermediate encapsulation protocol. As bindings exist to allow native communication between DCOM, CORBA, and Java, global open system communication between middleware-based active networks is

possible, regardless of the implementation specifics.

Many other benefits are gained through the use of middleware-based active networks. After the establishment of middleware interfaces for all packet processing routines, systems to support user-defined protocols within middleware-based active networks may be simplified. This is achieved by implementing user-defined packet processing routines as middleware components, which are then serialised, transported, instantiated, and invoked automatically by the middleware services of the host architecture. The component developer has the freedom to implement such packet processing components as desired, allowing end applications and users to request the services of such routines through standard middleware conventions, without any knowledge of the component's implementation details.

Middleware systems also offer extensive operating system support for the authentication of users and processes. A fine level of control over the registration, activation, invocation, and termination of packet processing routines is provided, a service that is core to any secure active network architecture. Additionally, the service of user process isolation is also common to middleware architectures, securing middleware-based active network architectures from client crashes. Memory protection is also provided by default, implying that client processes can only access data within their own address space. Any access to other data such as router state information must be authenticated, and invoked through well-defined interfaces.

As middleware components are applications in their own right, components loaded and invoked as plugin protocols from within middleware-based active networks have no limits on service possibilities. Middleware components contain their own data structures, and may request the same active network services that are available to external applications. As an example, packet processing components can query the active network routers for the current rate of throughput, and reroute packets according to QoS constraints. Intelligent caching, multicasting, congestion/admission control, mobile host support, NACK implosion protection, compression, and encryption are just a few of the many protocols that can be readily implemented as middleware components. Such components, when registered, can be utilised by an

application throughout the network with a single method call.

4.3.2 Existing Middleware-Based Active Networks

Several existing systems can be identified as middleware-based active networks. As a three year collaborative research project, FAIN represents an extensive middleware-based active network architecture, aiming to provide a highly configurable integrated hardware/software system [17]. The FAIN enterprise model has now been specified, with current work focusing on the development of an active node platform layer, a service programming environment, and a management system. As a component of FAIN, the Virtual Active Network (VAN) framework has been recently proposed [9], providing a generalised virtual private network, but with a greater degree of flexibility and control. Using middleware as the underlying technology, VAN will support resource partitioning and policing, packet demultiplexing and multiplexing, and cut-through links for packets that do not require active processing.

The ANTS active network can also be viewed as middleware-based, due to the inherent use of Java's distributed object communication model. ANTS does not, however, provide for open system communication and multiple language support. Finally, the RWANDA and TAO middleware frameworks also provide services that offer similar functionality to that of active networks. Regardless of their intended application, the findings of these systems provide valuable insight for the development of dedicated middleware-based active networks.

4.4 Architectural Design

COMAN has been designed as a lightweight yet practical desktop-to-desktop active network architecture, readily installable throughout a network of any topology, providing flexible packet processing and routing for next-generation services and applications. A key architectural feature is the high level of connectivity available to networked applications that utilise COMAN's services, and to other middleware-based active networks. COMAN is also designed to

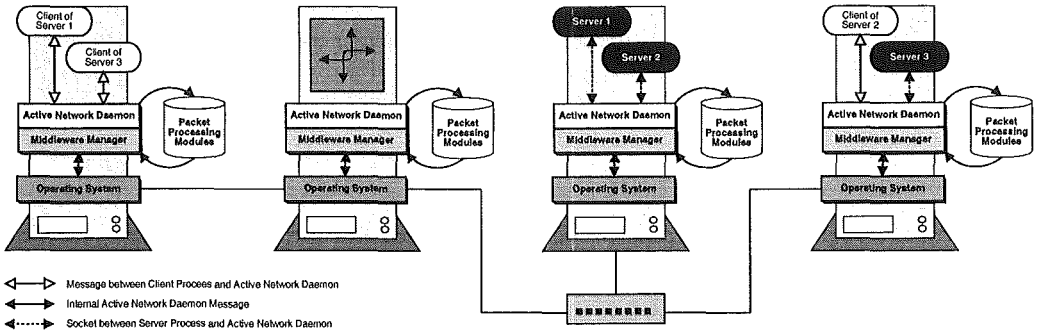

Figure 4.1: The high-level system architecture of COMAN.

enforce strong levels of user and process authentication.

All active networks aim to provide flexibility, efficiency, and security. However, considerable differences exist between COMAN and existing active network architectures. Whilst COMAN and FAIN share the common theme of middleware support, the COMAN architecture is considerably simpler, and is readily installable upon an unmodified operating system. Additionally, ANTS and COMAN both offer a high level of control over the processing of packets, but COMAN is not restricted by language, and provides a system-wide service. Finally, unlike CANES, SNAPd, and Smart Packets, an array of unrestricted languages are available through COMAN, with client process isolation mechanisms in place to enforce system safety.

4.4.1 System Features

The following paragraphs expand upon COMAN's key features, providing more details of the active network's architectural design.

Flexibility

User-defined packet and routing instructions, called *router modules*, may be developed in many languages, including C, C++, Java, Cobol, and assembly language. Router modules are implemented as middleware components that adhere to the common packet routing interface (see Section 4.4.4), providing a simple 'plug-and-play' mechanism for both module development and

invocation. No bounds are placed on the functionality of router modules, and applications may utilise the services of these modules through a single method call. COMAN provides control over the invocation of router modules; all data packets may be subjected to any given router module, or invocation of router modules may be restricted to all packets in a stream, or to individual packets. Packets may be subjected to processing from the sender to the receiver, the receiver to the sender, or in both directions.

Packet forwarding is related to operating system route tables, which allows the system to adapt to changes in the underlying route tables. For simplicity, COMAN uses conventional IP forwarding logic when routing packets, unless user-defined routing is requested. COMAN also supports any number of concurrent connections, providing a full forwarding service where packets are placed into a FIFO queue, processed, and then routed. Additionally, COMAN offers a simple API which closely emulates the popular BSD socket layer for the forwarding of data, with extra mechanisms to specify the invocation of router modules. Due to such socket layer emulation, client programs may be conditionally compiled as either an active or passive applications, using a source code preprocessor.

Practicality

COMAN executes on an unmodified operating system, entirely in user address space. No additional drivers are required, and the installation procedure is minimal, requiring no system restarts. Similarly, packet processing modules may be installed and deinstalled upon request, without interruption to the active network service. Additionally, it is not essential that COMAN resides on every network node. In cases where the active network service can not be installed, for example at a gateway router, COMAN will automatically revert to IP packet forwarding across the node. Subsequently, not even the endpoint nodes are required to have COMAN installed, although the existence of the active network service at endpoints will be beneficial in most cases.

As presented in Figure 4.2, COMAN is a layered architecture which provides independence from the underlying transport and network protocols.

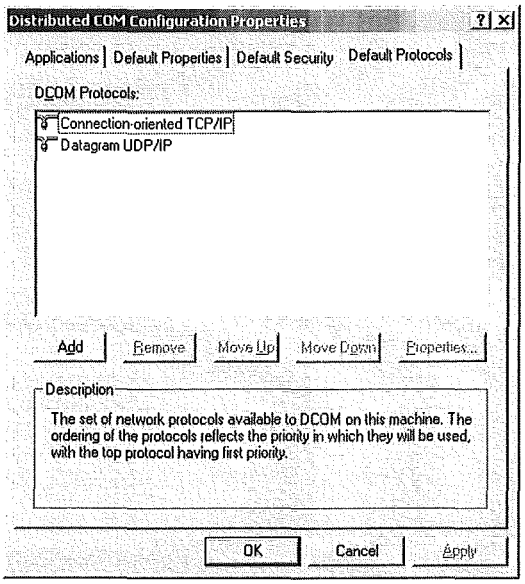

Figure 4.2: Configuration of the underlying COMAN transport protocols.

This allows COMAN to execute on any given transport protocol (or raw IP/IPX datagrams), presuming that the operating system network drivers are available. In the case of IPv4 to IPv6 protocol transition, the COMAN service can be reconfigured to use IPv6 without the need for service interruption.

Unlike previous active network architectures, COMAN does not require additional mechanisms for the explicit formation of packets, or for the custom marshalling of user data; these functions are implemented internally. As a core component of DCOM, the MIDL interface definition language compiler generates optimised RPC proxy-stub client code for the underlying marshalling of user data, allowing both the end-users and the COMAN system developers to avoid low-level coding routines.

Security

As a DCOM service, authentication of all active network components is maintained by the operating system, at both the user and machine level. Subsequently, it is possible to restrict the users that may connect to COMAN

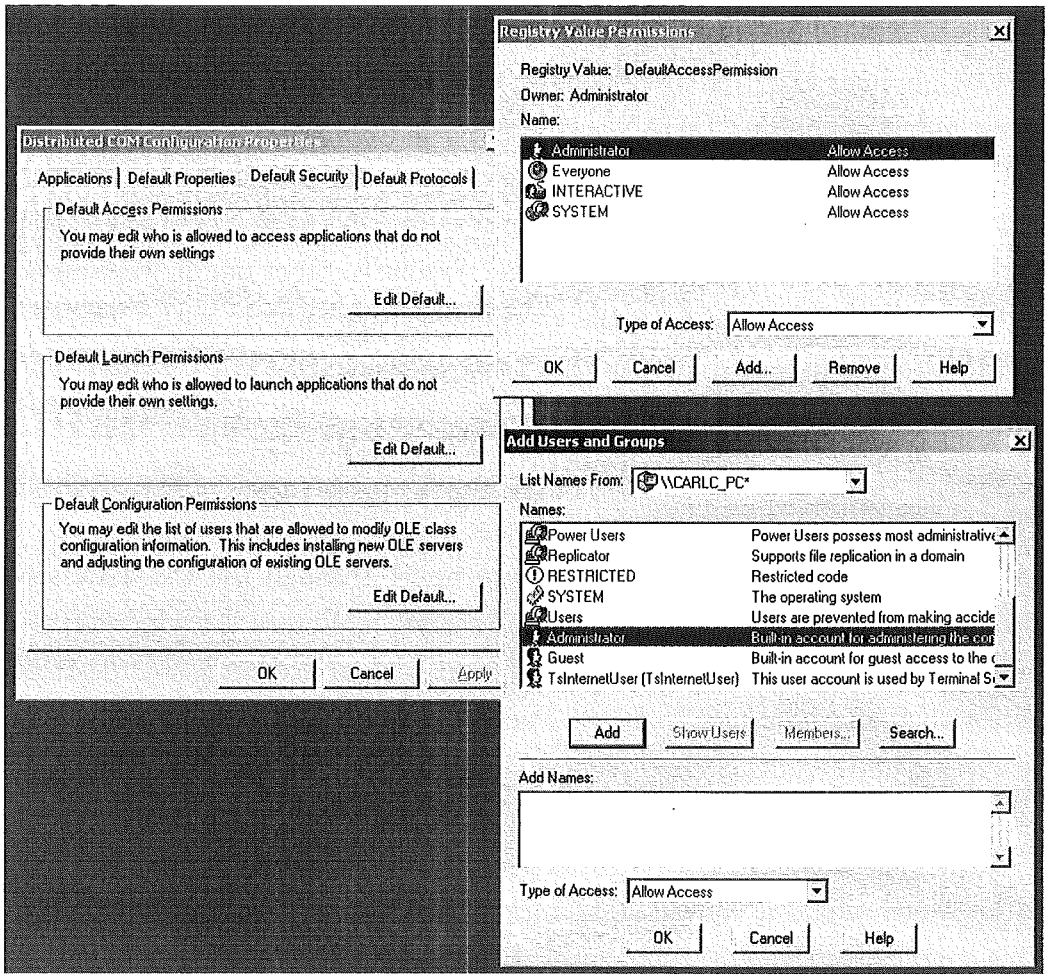

Figure 4.3: Built-in support for the specification of user access permissions.

and register, load, invoke, and unload router modules, as presented in Figure 4.3. If a malicious router module is executed within the network, only the data within the address-space of the local COMAN service can be compromised, as COMAN runs exclusively in safe user mode. Similarly, the COMAN architecture places each client in a separate process boundary, preventing unauthorised access to client memory, and isolating client crashes from other clients and the COMAN service.

Accessibility

DCOM bindings exist for C, C++, Java, and VB, allowing multiple-language access for clients of the COMAN active network. Additionally, several remote instances of COMAN active networks may communicate natively to form one logical active network, with the architecture reverting to conventional IP forwarding for all passive intermediate nodes. Similarly, other middleware-based active networks have the potential to communicate directly with COMAN, due to the middleware bindings that exist between DCOM, CORBA, and Java.

4.4.2 Service Establishment

Socket-based client/server streams may connect through the COMAN service resident at each active node, as presented in Figure 4.1. By default, the COMAN architecture forwards all data packets to the requested destination without payload modification. Packets may be subjected to additional processing and rerouting upon request through router modules, which encapsulate the specification of user-defined packet processing routines.

After attachment to the COMAN service, a client may specify a destination node where a conventional socket-based server process is listening. This commences the establishment of an *active channel.* At the source node, COMAN uses ICMP messaging to determine the next physical hop to the destination, based on the underlying route table. COMAN will then attempt to attach to the COMAN service running on the next hop, forming the first logical link in the active channel. This process continues until the destination node has been reached, at which point a local socket connection to the server process is created, and the active channel is established. In the context of the COMAN active network architecture, an active channel is similar to a conventional virtual circuit.

Upon establishment of an active channel, the client may begin sending and receiving data. When data is sent, it is placed into the local packet queue. The COMAN *worker thread* processes and routes each packet according to the instructions of the router modules associated with the channel. If a packet is not rerouted by any of the channel's router modules, the packet

is forwarded to the next active node in the channel's path. This process continues until the destination node is reached, at which point the data is delivered to the server process through the local server socket. Similarly, each active channel has a dedicated *listener thread* that is bound to the server process at the destination node. Upon receipt of data from the server socket, the listener thread places the data in the channel's received packet queue. Data may be removed from the queue through a blocking client-side method call, emulating the BSD `recv` socket layer routine.

4.4.3 Router Module Design

Router modules are implemented as COM components, which are then loaded by the local COMAN service at run-time. Router modules may contain instructions for the processing of packet payloads, which is an essential step when implementing encryption and compression protocols. Router modules may also contain instructions for the re-routing of packets, which is useful for the implementation of alternative routing protocols, such as multicasting and OSPF.

For router modules to be successfully loaded at run-time, they must implement the `IRouterModule` DCOM interface as described in Section 4.4.4. It is conventional to implement this interface using an object-oriented language such as C++, Java, or VB, although it is also possible to implement such interfaces using C, Cobol, or assembly language. By the virtue of middleware, the `IRouterModule` interface is decoupled from the implementation language, which allows the programmer great freedom when implementing router modules. Any constructs of the selected implementation language may be used, including arbitrary data structures (for storing state information), multi-threading, and interprocess communication.

The following is a code listing for a trivial router module, implemented in C++. The router module processes every packet that it is invoked upon, and does not re-route any packets. The router module simply replaces the first byte of data in each packet with the ASCII character 'A'. Whilst this is a somewhat meaningless router module, it demonstrates the relative ease in which a router module can be developed, and provides a useful outline for a

more sophisticated module.

```
class C_TestRMod :
  /* class C_TestRMod extends from a single-threaded wrapper class */
  public CComObjectRootEx<CComSingleThreadModel>,
  /* class C_TestRMod also extends from a globally-unique wrapper class */
  public CComCoClass<C_TestRMod, &CLSID_TestRMod>
{
private:
  CHAR          m_iData;  // private data for this router module
public:
  /* constructor which initialises data to the 'A' character */                10
  C_TestRMod()
  {
    m_iData = 'A';
  }

  /* Windows-specific macro which expands to give a complete class definition */
  BEGIN_COM_MAP(CRModT04)
    COM_INTERFACE_ENTRY(IRouterModule)
  END_COM_MAP()
                                                                               20
  /* IRouterModule methods */
  STDMETHOD(get_ReRoute)(INT * ReRoute)
  {
    *ReRoute = FALSE;  // return that we wont reroute for this example
    return S_OK;
  }

  STDMETHOD(get_Process)(INT * Process)
  {
    *Process = TRUE;  // return that we will process for this example          30
    return S_OK;
  }

  STDMETHOD(ProcessPacket)(BSTR * Data, INT * Len)
  {
    if (*Len > 0)
      {
        /* replace first byte of packet with this router modules current data */
```

```
      memcpy((LPVOID)*Data, &m_iData, sizeof(char));
    }
  return S_OK;
}

STDMETHOD(ReRoutePacket)
  (BSTR Data, INT Len, UINT Key, LPSTR Client, IReceiver * ActiveHost)
{
  return E_NOTIMPL;
}

STDMETHOD(get_Proceed)
  (UINT Key, LPSTR Client, IReceiver * ActiveHost, INT * Proceed)
{
  *Proceed = TRUE;  // dont halt the flow of data
  return S_OK;
}

STDMETHOD(get_Data)(INT ID, BSTR * OutData)
{
  return E_NOTIMPL;
}
};
```

4.4.4 The COMAN API

As presented in Table 4.1, the API for COMAN consists of two main interfaces. The `IComan` interface is returned to clients upon attachment to the COMAN service. This interface emulates conventional socket routines, provides support for the registration of router modules, and allows the retrieval of node and router module information. As an example of typical usage, Section 4.5 presents a simple C-based file transfer routine, demonstrating the creation of an active channel, the registration of a router module, the connection of the server socket, and the transmission of user data.

The `IRouterModule` interface specifies the routines that all COMAN-compliant router modules must implement. This interface is used internally by COMAN when routing packets through the network. Through the appropriate interface routines, COMAN determines which modules intend to

IComan		
AddManditoryModule	Send	GetModuleData
RemoveManditoryModule	SendTo	GetCurrentBPS
AddModuleToStream	SendUsingModule	CloseSocket
RemoveModuleFromStream	Recv	
CreateSocket	RecvFrom	
ConnectSocket	RecvUsingModule	

(a) The COMAN API for client communication.

IRouterModule
GetReRoutingMode
GetPacketProcessingMode
GetBlockingMode
GetData
ReRoutePacket
ProcessPacket

(b) The COMAN API for router module implementation.

Table 4.1: The core APIs for COMAN.

reroute and/or process packets, and invokes the processing and routing methods accordingly. As explained in Section 4.4.3, implementation of such routines is module-specific.

4.4.5 Implementation Details

As mentioned previously, COMAN is implemented as a DCOM service. The system executes in its own address space, and clients access the process through synchronous procedure calls which are managed by the operating system's DCOM libraries. The various address spaces are displayed in Figure 4.4, which presents a COMAN client/server active channel with associated router modules. In this example, the client process on Node 1 connects to the local COMAN service, which is in a separate address space. Com-

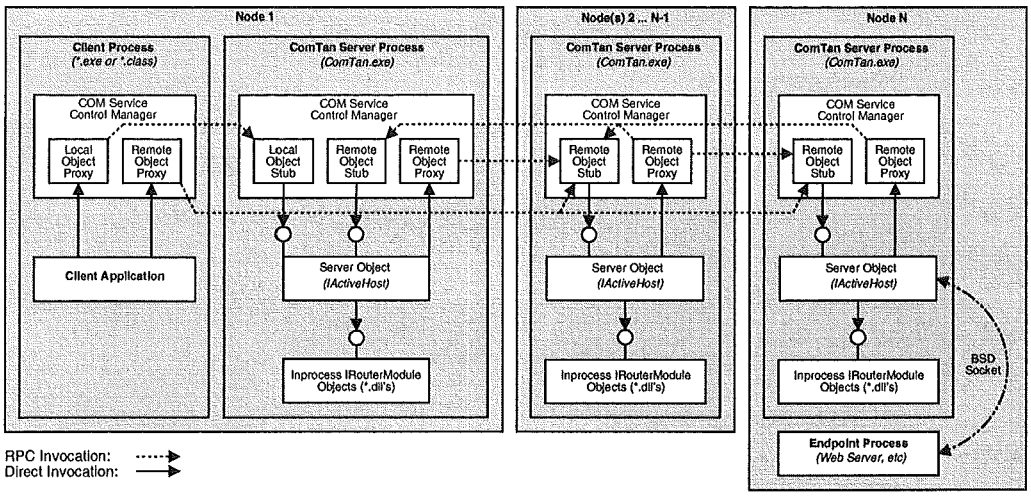

Figure 4.4: The address space layout of a client, the resident DCOM service, and a server process in the COMAN active network architecture.

munication is enabled via operating system maintained proxy/stub pairs, which are mapped into both address spaces by DCOM's service control manager (SCM). Additionally, this figure shows the address spaces of the loaded router modules on each active node. Finally, this figure also displays the remote server process at the terminating end of the active channel, which is again in a separate address space. Communication between the server-side COMAN service and the remote server process is enabled through standard BSD sockets.

Active channels represent client/server connections over the entire active network. Therefore, each active channel within the network must be uniquely identifiable. A combination of the server-side socket handle and the client-side machine name is used for channel identification. The COMAN service resident on each active node maintains a table of all active channels and associated router modules. Each packet in the network is stamped with the global channel identifier, and each COMAN service uses this identifier to determine which routing modules should be invoked when processing the packet.

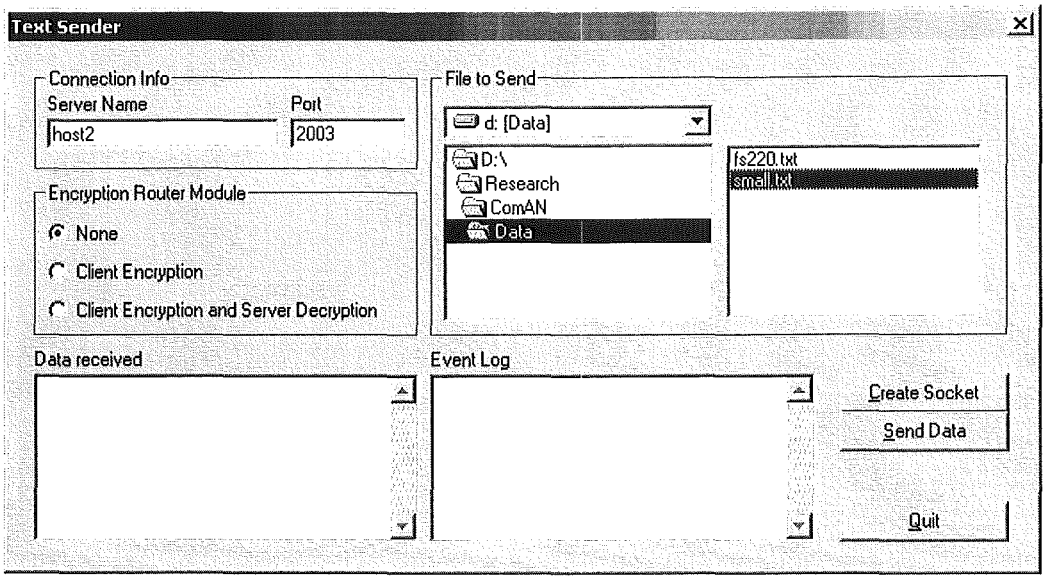

Figure 4.5: A COMAN client application that sends text files over an encrypted channel.

4.5 *Application of the COMAN Architecture*

The COMAN architecture may be used for any socket-based client/server communication. As presented in Chapter 5, multicasting, encryption, and file transfer router modules (and the calling applications) were implemented for the purpose of system evaluation. Router modules to support compressed data streams should also prove trivial to implement. It should also be relatively simple to implement router modules to provide support for congestion control, distributed data caching, dynamic routing, and QoS protocols. After a router module has been implemented to provide a protocol or service, it is uploaded to each active node throughout the network, where it is registered and access permissions are set, making the module available for all authenticated users.

Whilst all applications used during performance evaluations were written in C and/or C++, client applications can be written in an array of languages including Java and VB. Client applications that wish to use COMAN for client/server data transfer must link against the DCOM libraries. This allows

the program to attach itself to the running COMAN service and utilise the client-side API routines.

Once attached to COMAN, client applications may request the use of router modules through one of COMAN's router module loading routines. After requesting the use of all desired router modules, the client may then request to create and connect to the server process. Upon a successful connection, the client may then begin sending and receiving packets through one of COMAN's several send and receive routines, which emulate the BSD socket layer. At the end of typical client usage, the client disconnects from the server, and COMAN reclaims all allocated resources.

As a proof of concept, an example application written in VB is presented in Figure 4.5. The application sends a text file from the client node to the waiting server over an active channel. Once the active channel has been established, the user is allowed to select specific router modules which perform end-to-end encryption/decryption on the data stream.

It should be noted that COMAN permits connections to many types of server sockets. Most common are connection-oriented TCP sockets, but connectionless UDP sockets are also supported, as well as raw IP, which is typically used for ICMP messaging. Therefore, COMAN-based client applications can connect to all existing legacy server processes such as HTTP and FTP servers. Additionally, there are no limitations on the number of simultaneous client processes per active node—COMAN admission control is based upon a simple FIFO queueing model.

To provide an illustration of how a client may utilise COMAN, this chapter concludes by presenting a simple C++ file transfer application that connects to a waiting TCP-based server:

```
#include <windows.h>
#include "coman.h"

int main()
{
  IActiveHost *   p_IComan;     // pointer to the ComAN service
  SOCKET          SvrSocket;    // a local identifer for an active channel
  FILE *          p_File;       // a file to be opened for reading
  BSTR            bstrBuffer;   // a local buffer
```

```
/* open the local file */
p_File = fopen("./testfile", "r", 1);

/* attach to the ComAN service through a helper macro */
CREATE_INSTANCE(IID_ICOMAN, (LPVOID*)&p_IComan);

/* create a socket to the server process */
p_IComan->CreateSocket
  (
   AF_INET,           // use the IP address family                    20
   SOCK_STREAM,       // use TCP sockets
   "Server4",         // the name of the remote server
   SENDPORT,          // the listening port on the remote server
   &SvrSocket         // the active channel local identifier
   );

/* associate a router module with the active channel to perform encryption */
p_IComan->AddModuleToStream
  (
   SvrSocket,         // the active channel associated with the router module   30
   CLSID_ENCRYPT_MOD  // the global identifier of the router module
   );

/* connect the socket to the remote server process */
p_IComan->ConnectSocket
  (
   SvrSocket,         // the active channel to be opened
   TRUE               // boolean to bind the port prior to connection
   );
                                                                           40
/* read the open file, line by line */
while(fread((LPSTR)bstrBuffer, 1, BUFLEN, p_File))
  {
    /* send each line to listening server process */
    p_IComan->Send
      (
       SvrSocket,       // the open active channel
       bstrBuffer,      // the data to send
       BUFLEN           // the length of the data
       );                                                                  50
  }

/* close active channel now that file has been transmitted */
p_IComan->CloseSocket
  (
   SvrSocket          // the active channel to be closed
   );

/* close file handle */
fclose(p_File);                                                            60
```

```
  /* detach from ComAN service */
  p_IComan->Release();

  return 0;
}
```

Chapter V

COMAN: Architectural Evaluation

To provide an objective comparison in terms of latency and throughput for both single and multiple-hop data transfers, an empirical analysis of the COMAN and ANTS active network architectures was undertaken. Additionally, conventional C versions of test applications were trialled, providing a baseline comparison to passively networked applications. This chapter presents the findings from the above trials, and also presents a practical test case in which COMAN gained an increase in the end-to-end performance over conventional, passive networking.

5.1 Experimental Design

The performance of COMAN, ANTS, and passive C-based forwarding, in terms of throughput and latency, was measured on a switched 100 Mbps Ethernet network of six workstations and four routers, with the topology presented in Figure 5.1. Each workstation and router had a PC clone architecture, an ASUS 33 MHz single CPU mainboard, 128 Mb of primary memory, and an Intel Pentium II CPU with 16 Kb on-board cache memory operating at 233 MHz. All network cards were Intel Pro 100+Bs running at 100 Mbps in full duplex mode, and the operating system used throughout the network was Windows 2000 Advanced Server, Build 2195, Service Release 1.

The routers in the network consisted of multihomed workstations with static routes, with each router bridged by crossover Ethernet cables. All broadcast and background traffic was eliminated prior to testing. Finally, as ANTS was also tested on the network, Sun Microsystems' release of JDK 1.2.2 was used for compilation of the ANTS toolkit, and the Java 1.2.2-001 native threads virtual machine was used for interpretation.

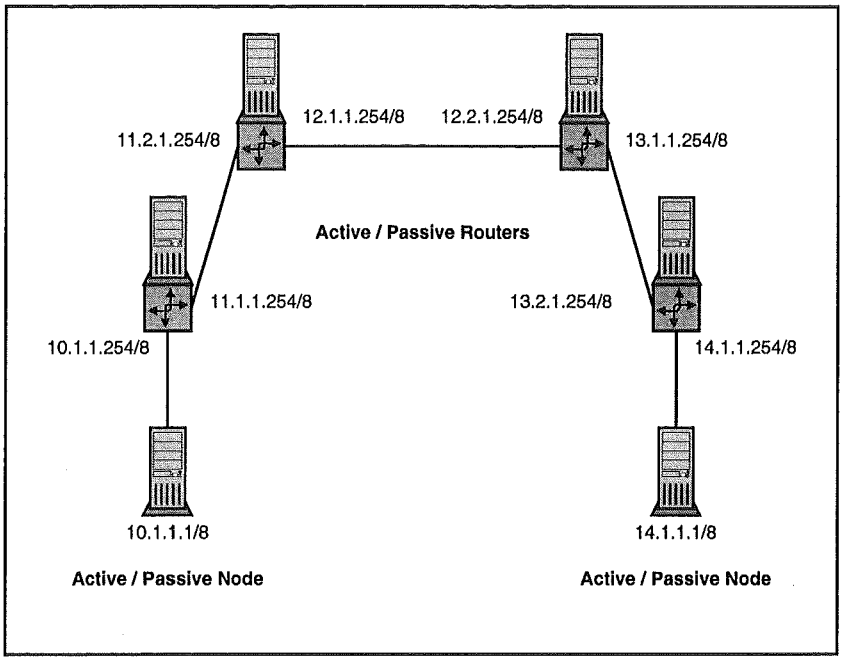

Figure 5.1: The network topology and link-layer configuration for evaluation of the COMAN and ANTS active network architectures.

All applications used during the testing period were initially written in C, and then translated to both COMAN and ANTS. The C version of the applications used standard IP routing (as supported by the kernel), giving close to the maximum possible performance. The active network-based applications emulated that of the C-based applications, with the exception that all processing was performed in user-mode, and then passed to the underlying network stack for hop-by-hop routing.

5.2 Results

This section presents various results from COMAN's performance analysis. Benchmarks in the form of throughput and latency are presented in Section 5.2.1, and an analysis of the impact of router module invocation is provided in Section 5.2.2. Applied performance results are also presented in Section 5.2.3, where COMAN was trialled in the context of multicasting and encryption services.

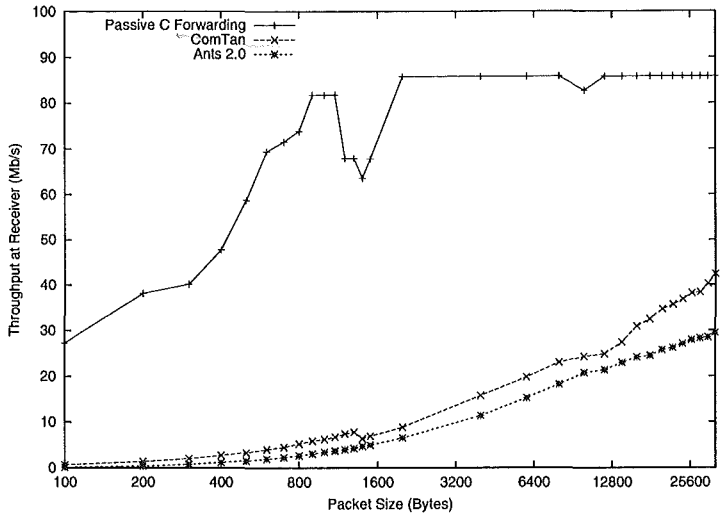

Figure 5.2: The average throughput of single-hop data transmission using passive C forwarding, COMAN, and ANTS with packet sizes of up to 32,000 bytes. Note the x axis is logarithmic.

5.2.1 Benchmarks

Throughput

The sustained end-to-end transfer of consecutive NULL characters was used to measure throughput rates. As a test mechanism, a TCP-based server process was bound at the destination node, where the server process would continuously accept data immediately after the establishment of a client connection. To determine the throughput rate achieved, COMAN established an active channel from the source node to the server process, and commenced the transmission of 200 Mb of data. The time taken to transmit the data was then used to calculate the transfer rate, which was then averaged over 30 trials. As comparative measurements, a passive C-based client and the ANTS active network architecture were also tested under identical conditions. The ANTS application developed to measure rates of throughput was translated as closely as possible from the relatively generic C and COMAN-based client/server programs.

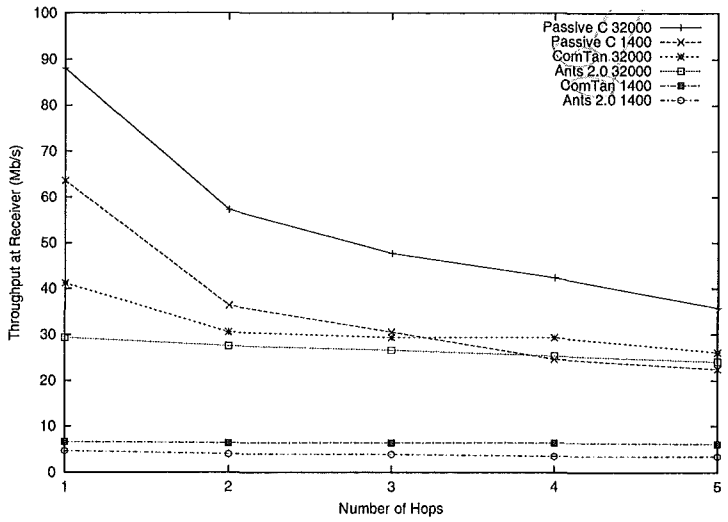

Figure 5.3: The average throughput of multiple-hop sustained transmission using passive C forwarding, COMAN, and ANTS with fixed sized packets.

Single-Hop Throughput The effect of packet size on the rate of throughput over a single-hop route is presented in Figure 5.2. This measurement provides an indication of the baseline throughput capabilities, without consideration of any active processing overheads that may be incurred by the active network architectures. The results show that C-based forwarding could achieve a throughput of 82 Mbps with a packet size as small as 1,000 bytes, yet even with a 32,000 byte packet size, COMAN achieved only 42 Mbps, followed by ANTS with 29 Mbps. Whilst only slight variance was experienced, all methods demonstrated performance degradation when packet sizes approached 1,500 bytes, due to link-layer fragmentation.

Multiple-Hop Throughput The effect of multiple-hop routes on the rate of throughput is presented in Figure 5.3. All three data transfer methods tested displayed varying degrees of performance degradation as the number of hops from sender to receiver increased. Whilst C forwarding appeared to be most affected by multiple-hop data transfer, the forwarding capability of the network was the actual limiting factor. The routers in the network are software based, and the hardware is of a relatively low specification, which

subjects C forwarding to confounded performance degradation as it reaches the throughput threshold for the network. The throughput rates achieved by the C forwarding did, however; provide an indication of the upper bound for the network's routing capacity.

The rate of throughput achieved by COMAN reduced from 42 Mbps for single-hop data transfers to 30 Mbps for multiple-hop data transfers using 32,000 byte packets. The 12 Mbps degradation in performance for multiple-hop data transfer represented the additional delay incurred by the active nodes that routed packets onwards, where packets were both delivered to and sent from the user-level COMAN service. An additional decrease in performance was experienced for five-hop data transfer using COMAN with 32,000 byte packets. However, again this is likely to be due to the limitations of the testbed network rather than a performance issue with COMAN. When reduced to 1,400 byte packets, both COMAN and ANTS demonstrated poor rates of throughput, with an approximate 85% deterioration in performance when compared to the throughput rates obtained using 32,000 byte packets. C forwarding's rate of throughput decreased on average by 40% when 1,400 byte packets were used.

Latency

To measure the latency of COMAN, a ping application was implemented. As comparative measurements, ping applications written in C and ANTS were also developed. The C-based version used ICMP messages, and for ANTS, a ping application was readily available from the ANTS 2.0 distribution. However, as the original ANTS version of ping had only millisecond granularity, the program was modified to call the high-precision timer as used by the C and COMAN applications. Identical to the throughput trials, latency was tested in terms of both packet size and number of hops between endpoints. The response times as presented in Figure 5.4 and Figure 5.5 represent the average of 200 consecutive pings, where times for each round-trip ping were halved on the basis that all delays within the network were symmetrical.

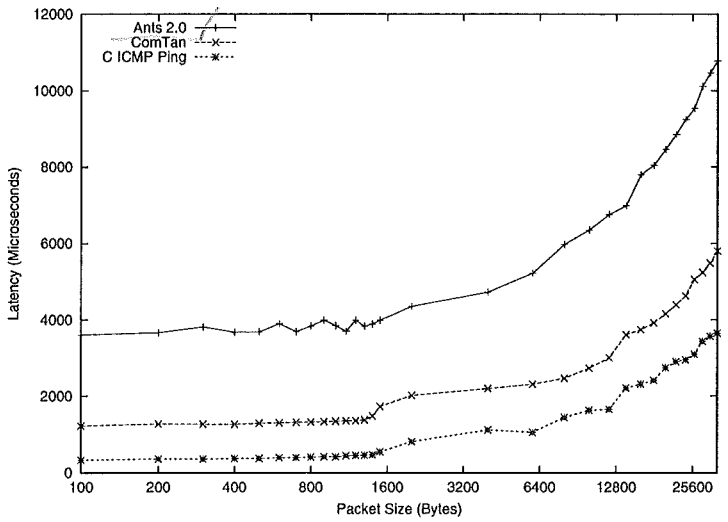

Figure 5.4: The average latency of a single-hop ping packet using passive C ICMP messaging, COMAN, and ANTS with packet sizes of up to 32,000 bytes. Note the x axis is logarithmic.

Single-Hop Latency As presented in Figure 5.4, all ping operations over a single-hop route displayed a linear increase in latency as the packet size increased. C-ping had a relatively low latency (ranging from $300\mu s$ to $3{,}600\mu s$), with COMAN-ping displaying approximately $1{,}100\mu s$ more latency regardless of the packet size used. ANTS-ping was considerably more latent, displaying at least twice the latency of COMAN-ping for every packet size tested. The large latencies incurred by ANTS are due to Java's bytecode-based virtual machine, which can not gain the level of responsiveness achieved by native code systems.

Multiple-Hop Latency Each version of ping displayed an increased latency according to the number of hops between endpoints, as presented in Figure 5.5. When using 1,400 byte packets, both C-ping and COMAN-ping doubled their latency when comparing response times from a single-hop to a five-hop route. At five hops, C-ping had an average latency of $1{,}400\mu s$, with COMAN-ping averaging $2{,}800\mu s$. However, ANTS-ping had increased its single-hop latency by a factor of five, with an average five-hop latency

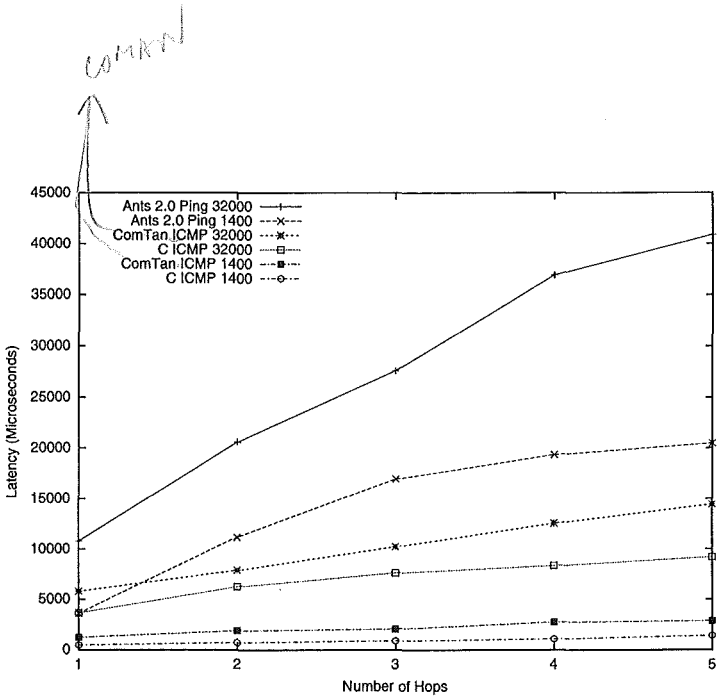

Figure 5.5: The average latency of multiple-hop ping packets using passive C ICMP messaging, COMAN, and ANTS with fixed packet sizes.

of 20,500μs. Using 32,000 byte packets, each version of ping approximately tripled its latency for five hops when compared to a single-hop route. For a five-hop ping, C-ping had an average latency of 9,000μs, COMAN-ping 14,000μs, and ANTS-ping 41,000μs.

5.2.2 Impact of Router Module Invocation

To measure the impact of COMAN router module invocation on the rate of throughput, multiple router modules per node were invoked on an active channel. The active channel was dedicated to the sustained transmission of client data, using the same throughput process as outlined in Section 5.2.1. As an easy-to-replicate test case, each router module simply set the first byte in each packet from NULL to the End-Of-File (EOF) character. This provided an unbiased measurement of the overhead related to the invocation of router modules, regardless of additional user-defined processing instructions. Rates of throughput were measured for routes of up to five hops from sender to receiver.

As Figure 5.6 presents, the invocation of a single module reduced throughput by up to 50%, regardless of the number of hops in the route. For the

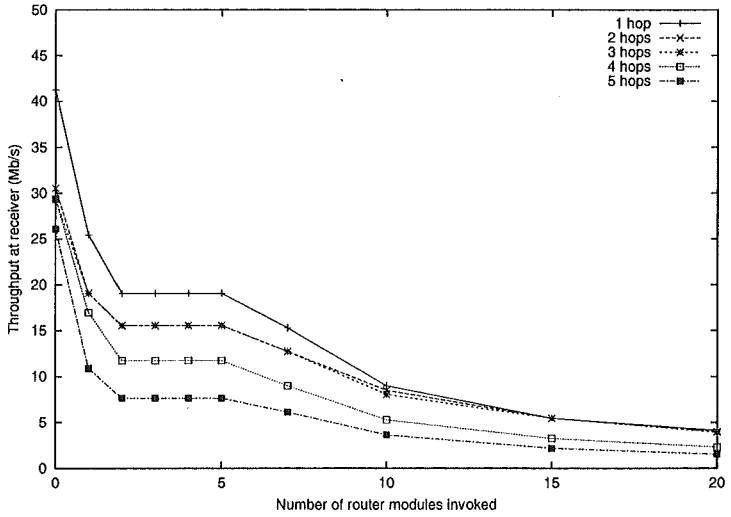

Figure 5.6: The impact of router module invocation on performance of sustained multi-hop data transfer for the COMAN active network with a 32,000 byte packet size.

invocation of between two to five modules, throughput remained constant, but at a deteriorated rate. As expected, deterioration in throughput increased with each additional network hop, due to the processing performed at each active node. For example, each packet of a single-hop/five-module active channel is processed ten times in total, whereas a five-hop/five-module active channel processes each packet thirty times prior to application delivery. As the results show, throughput rates continued to deteriorate as additional modules were loaded.

5.2.3 Architectural Application

Benchmark testing, such as the evaluation of throughput and latency, is useful for determining the base performance of a network. Active networks, however, are not intended to challenge conventional networks in terms of base performance—rather they provide tangible user-defined services in the absence of standardised solutions. The following sections present analyses of case studies where active networks have been used to provide new services on existing networks.

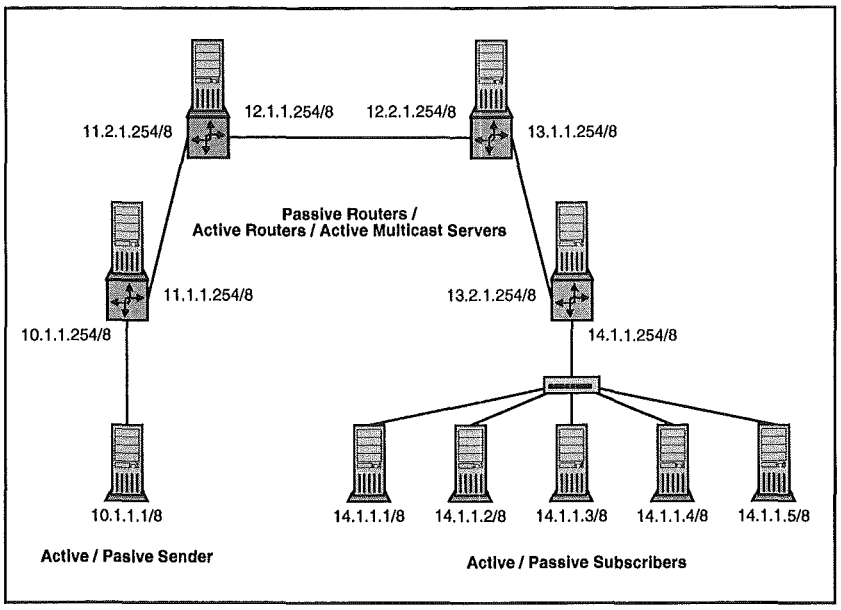

Figure 5.7: The network topology and link-layer configuration for evaluation of the COMAN and ANTS active network architectures.

Multicasting

A typical application of active networks is the emulation of multicasting services where no native multicast support exists. Accordingly, the throughput rates for COMAN and ANTS active network-based multicasting were evaluated, with results presented in Figure 5.8. The evaluation identified a scenario in which active networks provided throughput gains over conventional unicasting by reducing the number of redundant packet transmissions. The network topology used in the evaluation is presented in Figure 5.1. In all experiments, the sender (10.1.1.1) transmits multicast packets through four active routers, arriving at up to five receivers on the end subnet (14.1.1.*). The data transfer method used was identical to the sustained throughput method as outlined in Section 5.2.1, using 32,000 byte packets.

Four applications for the transfer of multiple data streams were trialled. A C-based unicast application was developed to provide standard unicast streaming. Multicast emulation applications were developed using COMAN

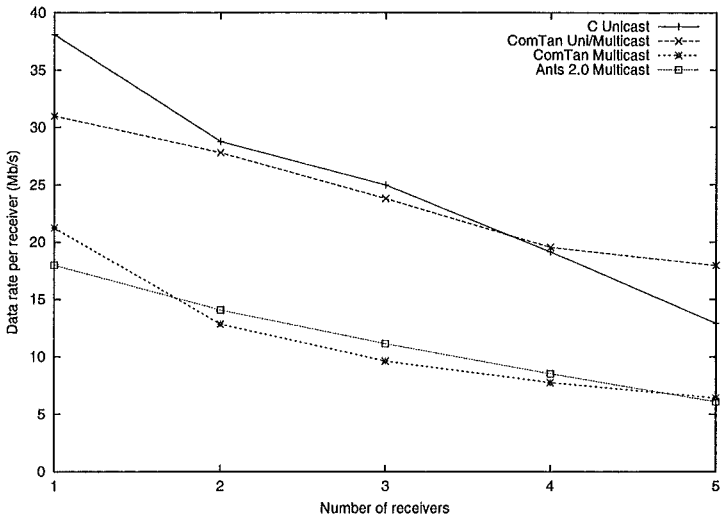

Figure 5.8: The throughput rates for unicast and emulated multicast data streaming with 32,000 byte packets.

and ANTS, where the exiting ANTS multicast application from the ANTS 2.0 distribution was modified to enable sustained data transfer. Finally, a hybrid application was developed using COMAN, aimed at providing improved throughput rates. The application established an active channel through the intermediate routers, but reverted to conventional unicast transfer between the end router and the receivers. This technique disallows any active processing at the receiving nodes, but vastly reduces the active processing requirements for the end router.

As Figure 5.8 presents, the C-based unicast application achieved throughput rates that were approximately twice as fast as the rates achieved by the COMAN and ANTS multicast applications, regardless of the number of receivers. However, for four or more receivers, the hybrid COMAN application outperformed the C-based unicast application, providing at least 18 Mb of data per second to each receiver. With five receivers, the burden of sending duplicate packets over the intermediate routers restricted the C-based unicast application to client throughput rates of 13 Mbps. It is likely that with a high-performance server architecture for the heavily loaded end router, COMAN's end-to-end performance would improve considerably, allowing a

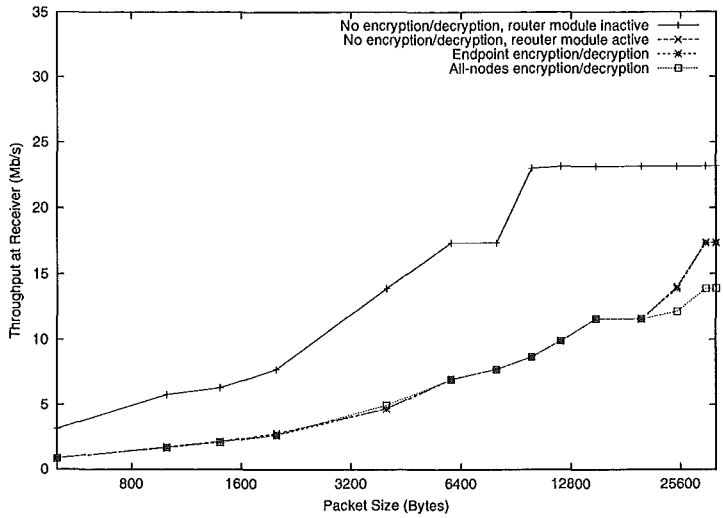

Figure 5.9: The impact on throughput for five-hop encryption using the COMAN active network with packet sizes of up to 32,000 bytes. Note the x axis is logarithmic.

full end-to-end active multicast service with throughput rates similar to that of the hybrid version.

Encryption

User-implemented encryption of data streams is a practical application of COMAN. To evaluate the impact of user-defined packet processing routines on throughput rates, an encryption router module was implemented. The router module provided *exclusive-or* bitwise encryption with a known key, where each byte in every packet was encrypted. Decryption consisted of reapplying the encryption step with the same key. An active channel was established to transfer a sustained stream of data to a listening TCP-based server process at the destination node, and the encryption module was invoked on all packets in this channel. A five-hop route was tested, with packet sizes ranging from 100 to 32,000 bytes.

To provide insight as to where router module processing overheads occur, various levels of module invocation were tested over distinct data streams. As a baseline comparison, data was transmitted without any router module

invocation. Data was also transmitted with a router module that did not perform any processing, in order to measure the overhead of per-packet router module communication. Next, data was transmitted with a module that would encrypt all packets at the source node, decrypt all packets at the end node, and inspect, yet leave packets unmodified, at all intermediate nodes. Finally, data was also transmitted with a module that performed decryption and re-encryption of data at all active nodes in the route.

As presented in Figure 5.9, user-defined processing had little direct influence on the deterioration of throughput rates, unless packet sizes were over 20 Kb. Rather, the invocation of the router modules, regardless of their user-defined routines, was the main cause of performance degradation. On invocation of a router module, each channel experienced an approximately 50% loss in throughput for most packet sizes, regardless of the level of user-defined processing specified by the router module.

5.3 Notes on Experimental Error

After presenting the experimental results, it is important to discuss the error associated with the data collected. For each of the experiments performed, only a negligible level of experimental data variation was observed. The lack of variation was due to the use of a fully switched testbed network which operated in full-duplex mode. The network configuration eliminated line contention at the Ethernet level, which made the observed rates of throughput and latency close to deterministic.

Other general notes about the experimental results can be made. As can be seen in Figures 5.2 and 5.4, the C version of each experiment ran at close to the wire speed of the network, as very little application-level processing was required during data delivery. Since the active versions of each experiment were CPU and I/O bound (and subsequently much slower than their passive counterparts), they were not significantly influenced by variations in the network's maximum possible throughput for the corresponding packet length. This 'smoothing' effect for active network throughput and latency was also observed during external analysis of the ANTS active network architecture [51, 50].

Another influence on the experimental results was IP fragmentation, which again can be observed in Figures 5.2 and 5.4. As explained above, the C version of each experiment ran at close to the network's maximum speed, hence the C version of each experiment was more affected by packet fragmentation than the active versions. This can be seen clearly at around the 1,500 byte packet length for all experiments. IP fragmentation was necessary on the testbed network as the link layer protocol used was Ethernet, where fixed 1,518 byte frames carried all higher-layer packets. The effect of IP fragmentation on active and passive networks is also displayed in the literature related to the PAN high-speed active network node [36, 35] (no other active network research has investigated the issue of packet fragmentation, or large packet lengths).

For packet lengths of approximately 64 Kb, another drop in performance is noticeable. Whilst this performance deterioration is currently unexplained, the most likely explanation is that this is a trait of the router operating system. This assumption requires further analysis to provide a justifiable theory.

5.4 Observations

During all trials, a positive linear correlation was present between the rate of throughput and packet size for both COMAN and ANTS. When large packet sizes are used, fewer active processing steps per byte are required; such packets are fragmented at the network layer and routed as a sequence of smaller sized frames at the link layer. This is similar to the technique employed by tag switching, where processing overhead per byte is reduced by diverting network layer-routing to link-layer switching after the identification of a packet stream.

Using low-specification desktop hardware, both COMAN and ANTS failed to provide rates of throughput capable of saturating a 100 Mbps Ethernet link. This implies that neither of the two active network architectures are currently suitable for use in an environment where high speed transmission is required, unless high-performance hardware is used. However, COMAN did provide relatively low latency, even with large packet sizes, indicating

that the architecture is well-suited to delay-sensitive, medium-throughput applications, such as realtime media streaming.

As indicated by the evaluation of throughput rates, COMAN's processing bottlenecks occurred at the active network routers. After profiling the active routers under a heavy loading, it was revealed that the number of CPU cycles consumed when copying data from kernel to user address space, and the time spent waiting for I/O routines to complete, were the main factors limiting COMAN's performance. By applying more processing power to active network routers in the form of multiple CPU machines and server architectures designed for high I/O utilisation, the performance threshold of COMAN is expected to increase throughout the network.

Finally, the development of specialised drivers could increase the performance of COMAN in terms of both throughput and latency, by directly forwarding packets in kernel-mode for cases where no active processing is required. Such drivers could also make other optimisations, such as modifying packet headers in kernel mode where possible, to avoid unnecessary context switches and data copies. At an even lower level, an alternative approach to reducing high I/O utilisation is in the form of programmable network interface cards (NICs), such as the *Arsenic* Gigabit Ethernet NIC [40]. The Arsenic 'connection-aware' NIC allows the uploading of packet filters from the operating system, providing limited connection-based packet processing without saturating the operating system with hardware interrupts.

Chapter VI

Conclusions

The active network paradigm provides networked applications and users with programmable interfaces that support the dynamic modification of a network's behavior, bypassing both the standards committee and the hardware vendor in order to cater for ever-changing user demands. Such flexibility is achieved by allowing the specification and invocation of complex processing instructions at all participating intermediate active network routers, facilitating the run time installation of arbitrary software-based routing protocols. Numerous active network architectures and components have been proposed by established research groups, with the collective goal to develop systems that address the key aspects of active networks: performance, flexibility, and safety. Whilst the concept of active networks is relatively new, several active network implementations have already been released for comment and further development.

With the continued growth in network-based applications, distributed middleware systems of various incarnations have emerged to manage issues of complexity and scalability. The theme of managed distributed object creation, remote method invocation, object persistence, and object termination is common to all distributed middleware systems, hence a close relationship exists between distributed middleware systems and active networks. Middleware systems offer many useful services for both the development of active networks and the integration of active networks with their host applications. As the primary result of this thesis, COMAN utilises such middleware services to provide arguably the most practical desktop-to-desktop active network architecture of today. Through the development of COMAN, the following observations have been made relating to a middleware-based active network architecture:

1. As an implementation platform, existing middleware services allow the rapid development of an active network architecture; mechanisms such as client authentication, out-of-process communication, and remote method invocation are available to be used as core architectural components.

2. As a client/server communication mechanism, middleware provides standardised interfaces for the binding of client applications, the active network service, and the end servers.

3. As a mechanism for the encapsulation of user-defined routing and processing instructions, middleware services provide a standardised, language-independent interface—decoupling user-defined packet processing routines from the underlying active network architecture.

COMAN introduces the concept of language-independent middleware components for dynamic service creation, and is readily installable throughout a workstation-based network, providing operating system supported user authentication and protection from malicious or erroneous system use. Scenarios have been identified where, through the use of such middleware components, COMAN improved the quality of application-based data transfers in terms of both performance and flexibility. Whilst evaluation showed that raw data transfer rates were slower than the rates achieved by conventional passive networks, considerable performance gains were made over the ANTS active network architecture. Additionally, the price paid in terms of performance for the functional COMAN architecture may be mitigated by more powerful router hardware or specialised drivers.

The multicasting and encryption modules, as presented during the evaluation of COMAN, demonstrate the architecture's practicality. The hybrid multicasting module provided a greater level of end-to-end throughput than conventional IP-based streaming for the congested network topology, and the encryption module allowed a different encryption realm over each hop in the network, with an acceptable decrease in throughput. Both of these modules may be freely distributed to active networks where such services are required, with the functionality accessible through a single call to the user API.

With a multiple-language active network now implemented entirely in middleware, future research aims to integrate COMAN with existing, specialised active network components for increased flexibility, safety, and performance.

6.1 Further Work

Whilst COMAN stands as a useful prototype, an important next step is the development of a standard set of middleware interfaces, allowing integration with other middleware-based active networks such as FAIN. Additionally, the development of an enhanced middleware framework that supports code verification is also a key area of future investigation. To increase safety and performance, COMAN, with the services of security-enhanced middleware, has the potential to support the SNAP packet language among others, providing an interoperable, efficient, and secure active network architecture.

The development and evaluation of QoS-specific router modules to assist the management of complex topologies is another potential project. Such work encompasses many research disciplines including protocol design and development, performance modelling and theoretical analysis, router module implementation and distribution, topology configuration and control, and end-application design.

Other work towards the COMAN active network could include porting the architecture to Unix. This is expected to be a trivial task, but if difficulties arise, the migration of the architecture to CORBA on both the Windows and Unix platforms would be an interesting research project. Additionally, COMAN currently has rather naïve admission control mechanisms which could be developed further.

An obvious, yet technically challenging, next step is the development of a router operating system that can support middleware-based active network architectures such as COMAN. Middleware systems are inherently layered on top of the network stack, hence one possibility is the migration of a full workstation operating system to router hardware. Unfortunately, this solution introduces performance and security issues, as highlighted in this thesis. The alternative is the development of a middleware-based router that has

full support for middleware services at the both the network and link layers, allowing active network architectures such as COMAN to be integrated with the router operating system.

Bibliography

[1] D. S. Alexander, B. Braden, C. A. Gunter, W. A. Jackson, A. D. Keromytis, G. A. Milden, and D. A. Wetherall. Active Network Encapsulation Protocol (ANEP). Active Networks Group RFC Draft, July 1997.

[2] D. Scott Alexander, William A. Arbaugh, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M Nettles, and Jonathan M. Smith. The SwitchWare Active Network Architecture. In *Communications Magazine*, pages 29–36. IEEE, October 1998.

[3] D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. Safety and Security of Programmable Network Infrastructures. In *Communications Magazine*, pages 84–92. IEEE, October 1998.

[4] Thomas Becker, Hui Guo, and Stamatis Karnouskos. Enable QoS for Distributed Object Applications by ORB-Based Active Networking. In Hiroshi Yasuda, editor, *The 2nd International Conference on Active Networks (IWAN)*, volume 1942, pages 225–238, Tokyo, Japan, October 2000. Lecture Notes in Computer Science Series.

[5] Steven Berson, Bob Braden, and Livio Ricciulli. Introduction to the ABone, June 2000. `http://www.isi.edu/abone/documents/ABoneIntro.ps`

[6] Samrat Bhattacharjee. On Active Networking and Congestion. Technical Report GIT-CC-96/02, Georgia Institute of Technology, February 1996.

[7] Jit Biswas, Aurel A. Lazar, Jean-Francois Huard, Loonseng Lim, Semir Mahjoub, Louis-Francois Pau, Masaaki Suzuki, Soren Torstensson, Weiguo Wang, and Stephen Weinstein. The IEEE P1520 Standards Initiative for Programmable Network Interfaces. In *Communications Magazine*, pages 64–69. IEEE, October 1998.

[8] Bob Braden and Michael Hicks. Active Network Overlay Protocol (ANON), December 1997. Active Network Working Group RFC Draft.

[9] Marcus Brunner, Bernhard Plattner, and Rold Stadler. Service Creation and Management in Active Telecom Networks. In *Communications of the ACM*, pages 55–61. ACM, April 2001.

[10] Maria Calderon, Marifeli Sedano, Arturo Azcorra, and Cristian Alonso. Active Network Support for Multicast Applications. In *Network Magazine*, pages 46–52. IEEE, May 1998.

[11] K. L. Calvert. Architectural Framework for Active Networks Version 1.0, July 1999. DARPA Active Network Working Group Draft.

[12] Kenneth L. Calvert, Samraat Bhattacharjee, and Ellen Zegura. Directions in Active Networks. In *Communications Magazine*, pages 72–78. IEEE, October 1998.

[13] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Structure of Management Information for Version 2 of the Simple Network Management Protocol. RFC, April 1993. SNMP Research, Inc., Hughes LAN Systems, Dover Beach Consulting, Inc., Carnegie Mellon University.

[14] Microsoft Corporation. The Component Object Model Specification. Draft Specification, October 1995. `http://www.microsoft.com/com/resources/com1598c.ps`.

[15] Kevin Curran and Gerard Parr. An Adaptable Quality of Service for Multimedia on the Internet. Distributed Multimedia Research Group, University of Ulster, February 2001.

[16] Darryl Dieckman, Perry Alexander, and Philip A. Wilsey. ActiveSpec: A Framework for the Specification and Verification of Active Network Services and Security Policies. In *Thirteenth IEEE Annual Symposium on Logic in Computer Science*, Indianapolis, Indiana, 1998. IEEE.

[17] Alex Galis, Bernhard Plattner, Eckhard Moeller, Jan Laarhuis, Spyros Denazis, Hui Guo, Cornel Klein, Joan Serrat, George T. Karetsos, and Chris Todd. A Flexible IP Active Networks Architecture. In Yasuda [53].

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[19] Erol Gelenbe, Ricardo Lent, and Zhiguang Xu. Design and Performance of Cognitive Packet Networks. *Performance Evaluation*, 46(2–3):155–176, October 2001.

[20] Object Management Group. The Common Object Request Broker: Architecture and Specification. Document 96-03-04, OMG, Framingham, MA and Reading, Berkshire, UK, June 1995.

[21] J. Hartman, U. Manber, L. Peterson, and T. Proebsting. Liquid Software: A New Paradigm for Networked Systems. Technical Report 96-11, University of Arizona, 1996 1996.

[22] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM International Conference on Functional Programming Languages (SIGPLAN)*, pages 86–93. ACM, 1998.

[23] Michael Hicks and Angelos D. Keromytis. A Secure PLAN. In *The First International Conference on Active Networks*, Berlin, Germany, June 1999.

[24] Michael Hicks, Jonathan T. Moore, D. Scott Alexander, Carl A. Gunter, and Scott M. Nettles. PlanET: An Active Internetwork. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM)*. IEEE, 1999.

[25] Cindy Kong, Perry Alexander, and Darryl Dieckman. Formal Modeling of Active Network Nodes using PVS. In *The Third Workshop on Formal Methods in Software Practice*, pages 49–59, Portland, Oregon, August 2000. ACM.

[26] Bobby Krupxzak, Kenneth L. Calvert, and Mostafa H. Ammar. Implementing Communication Protocols in Java. In *Communications Magazine*, pages 93–98. IEEE, October 1998.

[27] Fred Kuhns, Carlos O'Ryan, Douglas C. Schmidt, Ossama Othman, and Jeff Parsons. The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware. In *Proceedings of the*

IFIP 6 th International Workshop on Protocols For High-Speed Networks (PfHSN '99), Salem, MA, August 1999. IFIP.

[28] Ulana Legedza, David Wetherall, and Hohn Guttag. Improving the Performance of Distributed Applications Using Active Networks. In *Proceedings of the 17th Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM)*, San Francisco, California, April 1998. IEEE.

[29] William S. Marcus, Ilija Hadzic, Anthony J. McAuley, and Jonathan M. Smith. Protocol Boosters: Applying Programmability to Network Infrastructures. In *Communications Magazine*, pages 79–83. IEEE, October 1998.

[30] Doug Maughan. Active Network Research Web Site of the DARPA Information Technology Office. URL, September 1997. `http://www.darpa.mil/ito/research/anets`

[31] S. Merugu, S. Bhattacharjee, Y. Chae, M. Sanders, K. Calvert, and E. Zegura. Bowman and CANEs: Implementation of an Active Network. In *37th Annual Allerton Conference*, Monticello, Illinois, September 1999. University of Illinois.

[32] S. Merugu, S. Bhattacharjee, E. Zegura, and K. Calvert. Bowman: A Node OS for Active Networks. In *Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM)*, Tel Aviv, Israel, March 2000. IEEE.

[33] Jonathan T. Moore. Safe and Efficient Active Packets. Technical Report MS-CIS-99-24, Department of Computer and Information Science, University of Pennsylvania, October 1999.

[34] Jonathan T. Moore and Scott M. Nettles. Towards Practical Programmable Packets. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM)*, Anchorage, Alaska, April 2001. IEEE.

[35] Erik L. Nygren. The Design and Implementation of a High-Performance Active Network Node. Master's thesis, Massachusetts Institute of Technology, February 1998.

[36] Erik L. Nygren, Stephen J. Garland, and M. Frans Kaashoek. PAN: A High-Performance Active Network Node Supporting Multiple Mobile Code Systems. In *Proceedings of the 2nd International Conference on Open Architectures and Network Programming (OPENARCH)*, New York, NY, March 1999. IEEE.

[37] J. K. Ousterhout, J. Y. Levy, and B. B. Welch. The Safe-Tcl Security Model. `http://www.scriptics.com/people/john.ousterhout/`, March 1997.

[38] Gerard Parr and Kevin Curran. A Paradigm Shift in the Distribution of Multimedia. *Communications of the ACM*, 43(6):103–109, 2000.

[39] Larry Peterson. NodeOS Interface Specification, January 2000. DARPA Active Network NodeOS Working Group Draft.

[40] Ian Pratt and Keir Fraser. Arsenic: A User-Accessible Gigabit Ethernet Interface. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM)*, Anchorage, Alaska, April 2001. IEEE.

[41] Sathya Rao and Kaare Ingar Sletta, editors. *5th IFIP TC6 International Symposium, Internetworking 2000*, volume 1938, Bergen Norway, October 2000. Lecture Notes in Computer Science Series, Springer Press.

[42] Jens-Peter Redlich, Masaaki Suzuki, and Stephen Weinstein. Distributed Object Technology for Networking. In *Communications Magazine*, pages 84–92. IEEE, October 1998.

[43] J. Saltzer, D. Reed, and D. Clark. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems*, 2(2):277–286, November 1984.

[44] D. C. Schmidt. ACE: an Object-Oriented Framework for Developing Distributed Applications. In *Proceedings of the 6th USENIX C++ Conference on Object-Oriented Technologies and Systems*, Cambridge, Massachusetts, April 1994. USENIX Association.

[45] D. C. Schmidt, D. L. Levine, and S. Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324, April 1998.

[46] Beverly Schwartz, Alden W. Jackson, W. Timothy Strayer, Wenyi Zhou, Dennis Rockwell, and Craig Partridge. Smart Packets for Active Networks. *ACM Transactions on Computer Systems*, 18(1), February 2000.

[47] Jonathan Smith, David Farber, Carl A. Gunter, Scott Nettle, Mark Segal, William D. Sincoskie, David Feldmeier, and Scott Alexander. Switchware: Towards a 21st Century Network Infrastructure. `http://www.cis.upenn.edu/ switchware /papers/sware.ps`, 1997.

[48] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM JavaJust-in-Time Compiler. *IBM Systems Journal*, 39(1), 2000.

[49] David L. Tennenhouse and David J. Wetherall. Towards an Active Network Architecture. *Computer Communication Review*, 26(2), April 1996.

[50] David J. Wetheral. Service Introduction in an Active Network. Master's thesis, Massachusetts Institute of Technology, February 1999.

[51] David Wetherall. Active Network Vision and Reality: Lessons from a Capsule-Based System. In *17th ACM Symposium on Operating Systems Principles*, pages 64–79. ACM, December 1999.

[52] David J. Wetherall, John V. Guttag, and David L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *Proceedings of the 4th International Conference on Open Architectures and Network Programming (OPENARCH)*, pages 1–12, San Francisco, California, April 1998. IEEE.

[53] Hiroshi Yasuda, editor. *The Second International Conference on Active Networks*, volume 1942, Tokyo Japan, October 2000. Lecture Notes in Computer Science Series, Springer Press.

[54] Y. Yemini and S. da Silva. Towards Programmable Networks. In *International Workshop on Distributed Systems: Operations and Management*, L'Aquila, Italy, October 1996. IFIP/IEEE.

Glossary

ACTIVE NODE: A network layer device that performs active processing of packet payloads and user-specified routing.

ACTIVE PACKET: A network packet that is transmitted between active nodes. Such packets may contain user-defined instructions and/or user data.

CODE MODULE: See ROUTER MODULE.

CONTEXT SWITCHING: The task performed by the operating system when saving the state of the active process, and loading the saved state for another process.

EXECUTION ENVIRONMENT: The core components that collectively provide active packet processing at visible points in the network. Such components include the physical network host, a network stack, and an active network daemon/service.

KERNEL MODE: A mode of the operating system which allows the execution of privileged instructions. The operating system process and device drivers are executed in this mode.

PING: A method of sending a variable-length data payload between two hosts to measure latency. The most common ping implementation is ICMP ping, where link-layer support for ping requests has been standardised (see RFC 972).

ROUTER MODULE: A unit of code that contains the instructions for the routing and/or processing of network packets, also referred to as a *code module*. Router modules are located at active nodes and are invoked by the active network architecture upon user request.

USER MODE: A mode of the operating system in which direct execution of privileged instructions is disallowed. User applications normally are executed in this mode, providing address space protection from malicious or erroneous user code.

List of Acronyms

API	Application Programming Interface
ATM	Asynchronous Transfer Mode
BSD	Berkeley Software Distribution
COM	Component Object Model
CPU	Central Processing Unit
COMAN	Component Object Model Active Network
CORBA	Common Object Request Brokerage Architecture
DARPA	Defence Advanced Research Projects Agency
DCOM	Distributed Component Object Model
ELF	Executable and Linking Format
ICMP	Internet Control Message Protocol
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IPX	Internet Packet Exchange
MIB	Management Information Base
MIDL	Microsoft Interface Definition Language
NACK	Negative Acknowledgement
ORB	Object Request Brokerage
OS	Operating System
OSPF	Open Shortest Path First protocol

POSIX	Portable Operating System Interface for Unix
PVS	Prototype Verification System
QoS	Quality of Service
RFC	Request for Comment
RIP	Internet Routing Protocol
RPC	Remote Procedure Call
RSVP	Resource Reservation Protocol
SNMP	Simple Network Messaging Protocol
TCP	Transport Control Protocol
UDP	User Datagram Protocol
VTBL	Virtual Function Table

Appendix A

Code Availability

The source code for the COMAN architecture is available for download from `http:\\www.cosc.canterbury.ac.nz\~clc38\coman.html`. The source code is released for non-commercial use only, and includes all router modules and applications presented in this thesis. The ANTS applications and protocols presented in this thesis are also available from the above site.